Visual Basic

en.wikibooks.org

December 29, 2013

Contents

1 Introduction

1.1 How to use this book

The first page of the book is an annotated contents[1] list, click the headings with the mouse to go to the page you want to read.

- You will need a copy of Visual Basic. You can see the Microsoft US[2] or Microsoft UK[3] sites for information but the language has been retired and is now unsupported by Microsoft[4]. Visual studio 2005 is also available free of charge on the Microsoft website but only supports VB.net.
- You will often see Visual Basic abbreviated to **VB**, and for the purpose of conciseness, this guide will use VB from this point onwards.
- VB code is written in US English. There are some key differences you should note, as misspelling code can cause hours of problems. Below is a list of the differences- **when writing in VB, you should use the right-hand column's spellings**:

Many of these words you will not need until you progress into more complex VB

Commonwealth English	**US English (VB)**
Analyse	Analyze
Centimetres	Centimeters
Centre	Center
Colour	Color
Co-ordinates	Coordinates
Grey(ed)	Gray(ed)
Licence	License
Millimetres	Millimeters

Words which can be spelled with an 'ise' and 'ize' suffix should be spelled with 'ize' in VB.

This book will guide you step-by-step through Visual Basic. Some chapters of this book contain exercises. The exercises are an integral part of the text and at least some should be tried or you will not get the full benefit of the book. There are no answers to the exercises because most do not have a single *correct* answer. Physical exercises are done to build muscle and stamina, intellectual exercises are the same thing for the brain. In almost every

1 http://en.wikibooks.org/wiki/..%2F
2 http://www.microsoft.com
3 http://www.microsoft.co.uk
4 http://en.wikipedia.org/wiki/Visual%20Basic%23Legacy%20development%20and%20support

case you should be able to think of at least two 'solutions'. Some exercises require you to write code, please write it and execute it, others expect you to think about or describe something, please write it down even if only on a piece of scrap paper, it helps fix it in your mind.

If you try an exercise and fail to find a solution please ask for help by adding a comment to the relevant discussion page. It is of course entirely possible that the exercise is faulty so reports of problems will help the authors too.

Please add comments to the discussion page (click the *discussion* button at the top of this page). If there are specific topics that you want covered please say so in as much detail as you can.

1.2 Terminology

Please refer to the ../Glossary/[5] for definitions of any unfamiliar words or any words that appear to be used in an unusual sense.

1.3 VB Integrated Development Environment

Visual Basic has several key components you will need to get acquainted with. The first is the tool box, usually located on the left of the screen. This is used for creating objects on your form. A form is the basic window a user will see upon running your program. What is on the form is what the user sees. Make some picture boxes and command buttons on your screen by clicking the button, then clicking and dragging where you want the object to be placed. The next key feature is the properties of objects. Captions, names, and other properties of objects may be edited here. Now, click an object, and a white window will appear. This is where your code goes. Toward the top of your screen you should see a "play" button, but in Visual Basic it is referred to as the "Run" button. This will run any programs you make. Next to it are the pause, and stop buttons, which are pretty self explanatory.

5 Chapter 32.7.1 on page 234

2 History

Visual Basic is Microsoft's high-level object-oriented rapid application development[1] environment for the Windows platform. The first versions of Visual Basic were intended to target Windows 3.0 (a version for DOS existed as well), however it was not until version 3.0 for Windows 3.1 that this programming language gained large-scale acceptance in the shareware and corporate programming community.

Using drawing tools that resemble those found in hardcopy page layout programs or PhotoShop, VB programmers make user interfaces by drawing controls and other UI components onto forms. The programmer then adds code to respond to user interactions with the controls (for example, clicks, drag and drop, etc) known as events. The code can trigger events in other controls (for example, by displaying text or an image), execute procedures (run some algorithm based on the values entered in some control, output data, do business logic, etc), or almost anything else one might do in code.

Visual Basic can be considered to be an interpreted language like its Basic[2] ancestor, with appropriate modifications to accommodate object-oriented programming[3], and has implicit type conversion. That is, the VB development environment goes to great lengths to format (and aid the user in formatting) programming code so that it conforms to executable syntax. For example, VB will appropriately change the case of newly typed variable names to match those that have been declared previously (if they have been declared at all!). Traditionally, VB is known for compiling programs into pseudo-code (p-code, similar to Java's byte code) which is interpreted at runtime, requiring the use of dynamically-linked libraries (for example, VBRUN300.DLL for version 3 of Visual Basic, circa 1992) but newer versions can compile code into something more closely resembling the efficient machine code generated by C-like compilers. VB6 can be compile either into p-code or into native code; in fact VB6 uses the Microsoft C++ compiler to generate the executable.

For new Windows programmers, VB offers the advantage of being able to access much of the Windows UI functionality without knowing much about how it works by hiding the technical details. Although accessing low-level Windows UI functionality is possible, doing so in VB is as, or more difficult compared to such access using Visual C++ or other lower level programming languages. Recently VB.NET[4] has gone a long way to fixing some of the limitations.

Using custom controls provided by Microsoft or third parties, almost any functionality that is possible in Windows can be added to a VB program by drawing a custom control onto a form in the project.

---

1 http://en.wikipedia.org/wiki/rapid%20application%20development
2 http://en.wikibooks.org/wiki/Subject%3ABASIC%20programming%20language
3 http://en.wikibooks.org/wiki/Object%20Oriented%20Programming
4 http://en.wikibooks.org/wiki/Visual%20Basic%20.NET

Visual Basic traditionally comes in at least entry level and professional versions, with various designations depending on Microsoft's contemporary marketing strategy. The different versions are generally differentiated by the number of custom controls included, and the capabilities of the compiler. Higher priced packages include more functionality.

2.1 Evolution of Visual Basic

VB 1.0 was introduced in 1991[5]. The approach for connecting the programming language to the graphical user interface is derived from a system called *Tripod* (sometimes also known as *Ruby*), originally developed by Alan Cooper[6], which was further developed by Cooper and his associates under contract to Microsoft.

2.2 Timeline of Visual Basic

- Visual Basic 1.0 (May 1991) was released for Windows.
- Visual Basic 1.0 for DOS was released in September 1992. The language itself was not quite compatible with Visual Basic for Windows, as it was actually the next version of Microsoft's DOS-based BASIC compilers, Microsoft QuickBASIC compiler QuickBASIC and BASIC Professional Development System. The interface was barely graphical, using extended ASCII characters to simulate the appearance of a GUI.
- Visual Basic 2.0 was released in November 1992. The programming environment was easier to use, and its speed was improved.
- Visual Basic 3.0 was released in the summer of 1993 and came in Standard and Professional versions. VB3 included a database engine that could read and write Access databases.
- Visual Basic 4.0 (August 1995) was the first version that could create 32-bit as well as 16-bit Windows programs. It also introduced the ability to write classes in Visual Basic.
- With version 5.0 (February 1997), Microsoft released Visual Basic exclusively for 32-bit versions of Windows. Programmers who preferred to write 16-bit programs were able to import programs written in Visual Basic 4.0 to Visual Basic 5.0, and Visual Basic 5.0 programs can easily be converted with Visual Basic 4.0. Visual Basic 5.0 also introduced the ability to create custom user controls, as well as the ability to compile to native Windows executable code, speeding up runtime code execution.
- Visual Basic 6.0 (Mid 1998) improved in a number of areas, including the ability to create web-based applications using Internet Explorer. Visual Basic 6 is no longer supported.

5 http://en.wikipedia.org/wiki/1991
6 http://en.wikipedia.org/wiki/Alan%20Cooper

2.3 See also

- w:VBScript[7]

7 http://en.wikipedia.org/wiki/VBScript

3 Getting Started

3.1 Hello World

The very simplest program in VB is this:

```
Public Sub Main()
    MsgBox("Hello World!")
End Sub
```

To create this do as follows:

- Start VB
- A *New Project* window will appear. Select *Standard.exe* and click on *Open*.
- Open VB's *Project* menu and click *Add Module*
- Copy or type the Main subroutine shown above into the new code module.
- Go to VB's *Project* menu again and click the *Project1 Properties* item (it should be the last item in the menu).
- In the *Project Properties* dialog box you should find a dropdown list labeled *Startup Object*, click there and choose *Sub Main*
- Go to the *Run* menu and click *Start* (or press F5) to run the program. VB might ask if you want to save, just say no this time.

The running program will show the words **Hello World!** in a small message window.

3.2 The Visual Basic Integrated Development Environment

The Visual Basic **Integrated Development Environment** (IDE) is a program from Microsoft used to create, test, and deploy Visual Basic (VB) programs. VB programs can be written using other text editors, which must then be converted to executable code (compiled and linked) by the programmer. The VB IDE allows a programmer to write code and test it immediately, converting the source program to executable code as needed (on the fly). As the name implies, creation, editing, compiling, and linking are all integrated in this development environment.

The top of the IDE window shows VB's menu bar (see screenshot). Common menu categories (File, Edit, Help, etc.) as well as items specific to creating code (Project, Debug, Run) provide the text interface for the IDE. Commonly used menu items are provided as toolbar buttons directly below the menu bar.

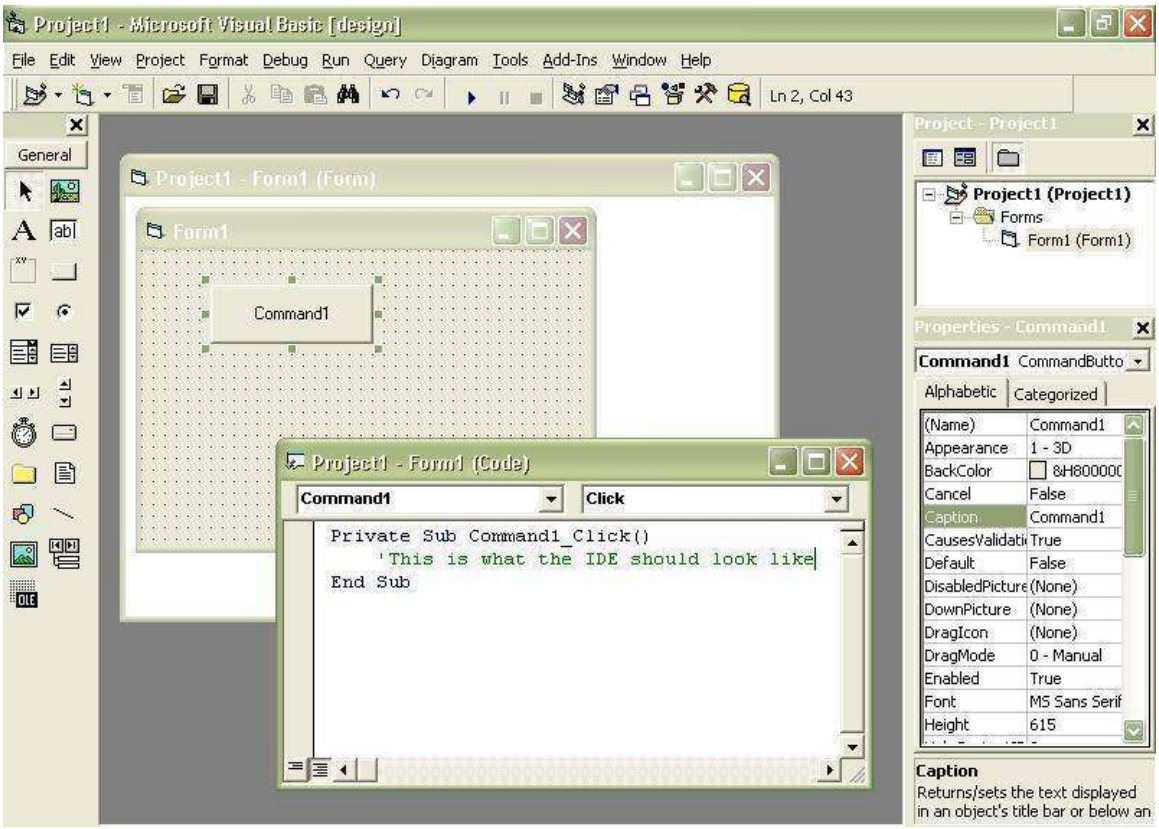


Figure 1 A screen shot of Microsoft Visual Basic 6 Integrated Development Environment

Two parts of creating a program; designing the user interface using forms and controls, and writing the source code; take up the majority of space in the IDE. As initially configured, four dockable windows also appear in the IDE's application space; the *Project Explorer*, *Properties Window*, *Form Layout Window*, and *Toolbox*. These and other supplied windows may be floated above the IDE, or docked to the sides, top, or bottom of the IDE application space as may be convenient for the programmer. Clicking on the title bar of a dockable window and dragging it around will dock, or undock, the window at will.

3.2.1 Availability

As of April 8, 2008, Microsoft no longer provides support to the Visual Basic 6.0 IDE.`http://msdn.microsoft.com/en-us/vstudio/ms788708/` Nonetheless, Visual Basic 6 is still available via MSDN subscription, under the name of "Visual Studio 6".`http://stackoverflow.com/questions/229868/how-to-compile-legacy-vb6-code`

3.2.2 Visual Editing

VB programs are divided into two views, the visual design of the user interface, and the code editors. The visual interface is created by adding controls from the *Toolbox* to the *Form Layout Window*. To add a control to a form; click on the desired control in the *Toolbox* and

then click and drag its bounding rectangle on the representation of the Form in the *Form Layout Window*. When you release the mouse button after the click and drag operation, the selected control will appear on the form.

Clicking on a control, or the form, in the *Form Layout Window* highlights the control (or form) with sizing handles. You can use the sizing handles to change the Left, Top, Height, and Width properties of a control or the Height and Width properties of a form. A highlighted control can be dragged to other positions on the form simply by clicking on the control and dragging it into place. A highlighted control can also be copy and pasted to reproduce additional identical controls on the form. Clicking and dragging around multiple controls (a lasso operation) will highlight them all as a group, which can then be moved as a group, or copy and pasted as a group, as may be needed.

3.2.3 Code Editing

Code editing is done using the code module windows. The title text of a code module window shows the project and module name. Two dropdown lists show the objects accessible to that module (General) and the subroutines (Declarations) associated with the selected object in the General list. In a form's code module, the General list will list all of the controls that have been placed on the form. Each control has its own set of events and methods, which will populate the Declarations list upon selection of any control. These two windows combined help the programmer locate specific code for events and methods in the program. In small programs, the editors can be scrolled up and down to find a desired routine to work on. In large programs finding the right spot can be much more difficult, where these two lists bring the programmer to the desired location using a few clicks of the mouse.

The code editors will show the program as one long text document, or it can show the program a single routine at a time. The two buttons in the lower left of the code module window allow the programmer to set his/her view preference.

The code editors can be split into two views by moving the small bar in the upper right of the editing area. The two views show two different views of the **same code module**. Again, in larger programs, it may be helpful to see one section of the code in the module while working on another area of the code in the module. The split view mode allows the programmer to keep one section of the code in view while creating a new method or function.

3.2.4 VB Project Files

Code modules and forms are saved individually as text documents in the project folder. The filename extension (.xxx) is used to identify which type of module the file contains. The project file (.vbp) describes the project as a whole, naming the individual files needed to create the program. With VB installed correctly, double clicking on a project file will bring the entire project into a new instance of the VB IDE, while double clicking on a form (.frm) or module (.bas) or class (.cls) or control (.ctl) file will load that individual file into a new instance of the IDE. Typically, loading a single file from a multi file project (other than the project file) will not have all the information required to allow the program to run. However, in many cases, short test code, code written to test out some small idea, is best

kept in its own module (and file) where it can be loaded and discarded at will. Such code can be added, removed, or edited, until it is perfected (debugged) and copied into the real program modules.

3.2.5 Immediate Window

While testing a program for correctness (usually called debugging) it is often advantageous to see interim results in a calculation, or to check on values being passed to a subroutine. Setting breakpoints to stop the program at specific lines can help, as can sending printed values to a second window. VB provides a dockable window for text called the **Immediate Window**. It is available under the View menu or you can press CTRL+G to bring it into view. It is called the Immediate Window because it can be used to execute BASIC commands immediately. For example:

```
Print Atn(1) * 4
```

When you type that line into the Immediate Window and press Enter, VB interprets that line as a VB command and executes the command. In this case it prints the number 3.14159265358979 on the line following the command. (For expediency, the question mark can be substituted for the Print command in most circumstances.)

You can send text to that same window from a running program using the Print method of the Debug object:

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    Debug.Print Chr(KeyAscii), KeyAscii
End Sub
```

In this case, with a Textbox called Text1 on some form, every letter a user enters into the textbox will be echoed to the Immediate Window, as well as its numerical ASCII value.

Because text is sent to the Immediate Window using the Debug object, many people refer to the Immediate Window as the Debug Window. You may find it listed either way in this literature.

IDE NOTE:

It should be noted that running a program using the VB IDE is not *identical* to running a stand alone executable program. The program running inside the IDE is running as a sub-process of the IDE. There are a few small connection and timing issues associated with running as a sub-process. On the whole, however, a program run in the IDE will look and feel like the finished program, but to fully test a program as the user will see it, it is advisable to create the program executable (.exe) and test that for correctness and usability.

3.3 Parts of a Visual Basic Program

3.3.1 Forms

A form is a window where controls are placed for use by the user of the program. The .Caption property changes the text on the title bar of the window, and the .MinButton and .MaxButton properties show or hide the minimize and maximize buttons. Different window styles such as Dialog boxes, ToolWindows, and standard Forms; as well as some allowed user actions such as resizing and minimizing; are controled by the form's .BorderStyle property. It is a common practice to name a form with *frm<FormName>* (ex: frmMain, or frmAlert).

3.3.2 Components

A component is an executable module stored either as a .VBX file (Visual Basic eXtension for the 16-bit versions of VB), .OCX (OLE Control eXtension for 32-bit versions) file or as a .DLL (Dynamic Link Library) file. Components are pre-compiled code modules used by other program writers with/without knowledge and understanding of the details of its inner workings. Pre-compiled components provide reuseable code that has already been written and debugged. Components can be code only (.DLL) or have a visual component that can be placed on a form (.VBX and .OCX). VB supplies many commonly used components (Button, Textbox, Listbox, etc.) as controls in the *Toolbox*.

3.3.3 Events

An *event* is an activity that occurs during a program's execution usually in response to the user's actions, such as a mouse click or a keypress. An event causes a *procedure* to execute. You the programmer can decide what code (if any) you want to place within the different event procedures.

3.4 Controls

3.4.1 Control Properties

To display the Control Properties window, select View\Properties Window or press the F4 key. The Properties window initially appears on the right edge of the main window and contains all the names of the editable properties as well as their current values. Some control properties are only editable while the program is running, some are only editable while in design mode.

3.4.2 Buttons

A button will be your best friend in Visual Basic. Each button should contain code, which is added by you, the programmer. Upon clicking the button, the user will be instructing the program to execute that portion of code. For example, you could set it so when pressed, the

program will make a message box that says "HELLO!". Good programming styles generally use *cmd<ButtonName>* when naming a button.

3.4.3 Text boxes

Text boxes allows the users to add text areas to their programs. This text does not have to be typed in directly by the programmer, but could come from other sources such as database fields, text files or data the user will type in while the program is running. Although the default value for this is the name of the control, it can be set to anything including "" (or nothing). Text box names are prefixed with txt, eg; *txt<BoxName>*.

3.4.4 Labels

Labels are one of the most used Visual Basic objects. They are often used to label other controls (textboxes, images, etc.) or provide feedback to the user. They are usually named like *lbl<LabelName>*.

3.4.5 Timers

Timers are interesting and easy to learn. If you want the program to perform a certain task after a certain amount of time, the Timer is there to help you out. Their only event procedure is _timer, which will be executed every time after a certain amount of time is passed. The most common steps to use Timers is as simple as follows:

1. Add a timer to the form and give it a name.

2. Set the time interval in the *Properties* window to some value above 0.

3. Double click the timer and add the code you want executed at the set intervals.

Timers have very few properties too.

This is a possible use of timer: (To see it in action add a Command button, Shape control, and Timer to a new form. Then set the Timer.Interval property using the *Properties* window. Copy the code below into the form's code module and press F5 to run the program.)

```
Private Sub Command1_Click()
    Timer1.Enabled = Not Timer1.Enabled
End Sub

Private Sub Timer1_Timer()
    Shape1.Visible = Not Shape1.Visible
End Sub
```

This would make the command button stop and start the flashing (repeatedly disappear and reappear) of the Shape control. The flash interval is determined by the Timer's Interval property. **Timer.Interval** is measured in milliseconds.

Here is another timer example of a counter:

```
'************************************************************** *
'APRON TUTORIAL PRESENTED BY MORROWLAND *
'************************************************************** *
'Project Name : Timer * * Project Description : Using Timer and
'Counter * * Project Type : Visual Basic * * Author : Ronny André
'Reierstad * * Web Page : www.morrowland.com * * E-Mail :
'apron@morrowland.com * * Version : English (UK) * * Date :
'27.06.2002 *
'**************************************************************
'Timers are the backbone in any good application, you will be able
'to decide when things will happen in millisecounds by using timers
'and counters you gain control

'declare counter as integer
Dim counter As Integer


Private Sub Form_Load()
    Timer1.Enabled = False 'disable timer at startup
End Sub


Private Sub Command1_Click()
    Timer1.Enabled = True   'starts the timer by enabling it
End Sub


Private Sub Command2_Click()
    Timer1.Enabled = False  'stops the timer by disabling it
End Sub


Private Sub Command3_Click()
    counter = 0             'reset the counter
End Sub

'The timer procedure
'the timer procedure will loop in the interval of the timer
'I have set the timer interval in the "properties" menu to 1000 ms (1 sec)
Private Sub Timer1_Timer()

    counter = counter + 1   'we set the counter to count here

    Text1.Text = counter    'write the counter value out as text

End Sub
```

VB Timers are given low priority on the processing totempole. If any other code needs to execute when the timed interval has elapsed, that other code is allowed to execute before the timer's procedure is called. Other code can yield some of their processing time to Windows by using the DoEvents command. That is the only VB command that will allow code you write to yield time to any pending timer events in the program. Without the use of DoEvents, **each subroutine or function is executed start to finish, in serial**

fashion. In other words, only one command or statement can be executed at a time. Each subroutine or function must list the proper commands to execute, in their proper order.

3.4.6 Picture boxes

Although called picture boxes, these objects are not just a heavyweight version of image boxes: picture boxes almost have the same properties and function as Form objects. It can do far more than just displaying pictures. Probably the best way to describe picture boxes is that they are containers that can group other objects together, kind of similar to frame objects. E.g. several command buttons can be drawn "inside" of it.

See also ../Simple Graphics/[1].

3.4.7 General properties

VB extends some common properties to all controls placed on a form. Name, Top, Left, and Tag, are a few of the extended property names. When you lasso and highlight several controls on a form, you can change the general property values of all the highlighted controls using the *Properties* window. Making a change there will change the property of the same name for all of the highlighted controls.

3.4.8 References

Reference is a kind of link that is used by Visual Basic. The word reference is used in two distinct ways:

Component reference

Are typically external libraries such as DLLs[2], OCXs[3], or type library[4]. Component references are added to the project, so that all parts of the program may interact with the outside component.

Object reference

Is typically a pointer to an internal object[5]. Object references differentiate, or identify, specific objects within the program. The term object is a very general term used to indicate just about anything that has properties to set, or methods to call. For example, if your form has 5 textboxes on it and you want to put text in just one of them, you need to use the proper 'object reference' to address that specific textbox.

These two uses are quite distinct and generally do not cause any problems. Usually when *object reference* is meant it will be written out in full whereas just *references* usually means references to an external library. In both cases, accessing the properties (or methods) of a

1 Chapter 17 on page 91
2 http://en.wikibooks.org/wiki/..%2FDLL
3 http://en.wikibooks.org/wiki/..%2FOCX
4 http://en.wikibooks.org/wiki/..%2Ftypelib
5 http://en.wikibooks.org/wiki/..%2Fobjects

component or object reference requires a special syntax that includes a period to separate the object and property. The syntax is: .<property> or, by example: Form1.Caption = "This", or Text1.Text = "That", etc. where Form1 or Text1 are the specific objects you want to alter, and Caption or Text are the specfic properties being changed.

You can create components[6] in VB to be used by other programs (not just those written in VB).

3.5 Reserved Words

Visual Basic contains several reserved words. These words are "reserved" because they are specific functions and commands in Visual Basic. For example, a variable may not be named "Print" because it is a feature in VB to print. This can be avoided however, by naming your variables "prnt" or "print1". As long as it is not the exact word, it should work. A list of frequently used reserved words/keywords:

```
And
As
Beep
Call
Close
Command
Date
Do
End
Error
Event
Exit
False
For
Function
Get
GoTo
If
Input
Kill
Let
Load
Loop
Me
Name
Next
Not
Nothing
On
Option
Or
Print
Private
Public
Put
Reset
Resume
Set
Step
```

6 Chapter 3.3.2 on page 13

```
Stop
Sub
Then
Time
True
Until
While
With
```

3.6 REMs

While programming you may find it necessary to leave yourself notes. This can be used to easily identify areas of code, or as a reminder of sections of code with logic errors that need to be fixed. REMs are very simple. Merely place an apostrophe, " ' ", or the word, "REM", before the line and that line will be ignored by the interpreter, whether it be a message to yourself or a section of code. For example:

```
' I leave notes to myself
REM or I could do it like this
'If I REM a line of code it will not execute!
REM x=5
```

An apostrophe is most often used due to the comparative reduction in space and time as REM has been included so as to be backward compatible with earlier versions of BASIC. The REM statement is a basic command and must be preceded by a colon if following other commands on the same line. An apostrophe needs no colon:

```
x = 5 'This comment will work
y = 3: REM and this comment will work.
z = 1 REM This line would cause a syntax error.
```

3.7 Error Checking

These are several common types of errors that one might get from a VB program:

run-time errors

This type of error are errors that are raised when the program is running. Examples: dividing anything by zero or assigning a string to a numeric variable.

compile errors

Compile errors are errors that the IDE spots at design-time and upon compiling the program just before it runs the program. These errors include syntax errors -- the error raised when the computer does not understand your code, and also errors like undeclared variables etc.

logic errors

Logic errors are errors that the computer cannot spot. These are errors that, even though the program runs, the result is not what you intended.

The first two of these errors are generally easy to spot, and the debugging tool can be used to high-light the line of text at which the error occurred. For example, if you wanted to make a program to convert Celsius to Fahrenheit, and in the code you used a multiplication symbol instead of division, the program would run fine, but the program would not convert the temperatures correctly, as you desired. Although sometimes these logic errors can be easy to spot, some are quite difficult. Logic errors become concealed in the lines of code when making complex programs, such as a game. Just remember, the computer does not know what the program is supposed to do, it only knows the code. Look through the code step-by-step and think of how the computer would interpret the code.

VB won't let you just ignore syntax/compile errors. You have to fix them before you can run your program. But run-time errors, which are syntactically correct, but may cause an error if an attempt is made to execute it, can be handled to possibly prevent your program from crashing. The following example shows a very good way of handling a possible error:

```
Private Sub Form_Load()
    On Error GoTo ErrorHappened
    i = 1 / 0 'This line will cause an error to be raised as anything divided
by zero = infinity
    '...
    'Some code
    Exit Sub 'Here the subroutine is exited if no errors occur
ErrorHappened:
    'Informs the user an error happened.
    MsgBox "Error Number " & Err.Number & ":" & Err.Description
End Sub
```

The output of this example is a message box that says "Error Number 11: Division by zero". The statement *On Error Goto ErrorHappened* will skip everything from `i = 1 / 0` to `Exit Sub` if any run-time error occurs within this procedure. When an error is detected the program will continue to run from right after *ErrorHappened:*, where the error will be displayed so that the programmer can find out what it is and fix it. This also prevents the program from "crashing".

ErrorHappened is just a line label, you can name it anything you wish to identify that section of the procedure you want to execute when an error happens. *On Error Goto* can only reference line labels that are within the current procedure. You cannot (easily) define one error handler for all procedures. Exit Sub means to end the Form_Load event immediately.

So if no error occurs, a message box will NOT be called because *Exit Sub* will already have ended or exited our subroutine. And if an error does occur, the message box will pop up, displaying the Error Number and the Error Description.

The above example is the safest way of detecting and handling any error that takes place in the subroutine. However you can also choose to ignore errors by using "On Error Resume Next" which means to ignore all errors. Ignoring errors is not a good idea in most scenarios, but in some cases the proper way to handle an error is to do nothing, so they might as well be ignored. Even using Resume Next, you can test the Err object (Err.Number) to see if an error happened on preceding lines.

See Errors[7] for more detail about error handling.

3.8 Declaring Variables (Dimensioning)

If you don't already know, a variable is, by dictionary definition: a symbol (like x or y) that is used in mathematical or logical expressions to represent a variable quantity. In mathematics, common variables are: x, y, z, etc., and they can "hold" values like x=1, y=3, etc. In VB, instead of x, y and z, a variable can have whatever name you want. It is often good practice, and sometimes necessary, to dimension variables. Often it is called 'dimming'. This process gives the variable its name and the type of value it will be able to hold (which will be discussed later). To dimension a variable, the code is:

```
Dim variablename [As Type]
```

Of course, the variable name could be whatever you want. The type however, is different. You have a choice of single, integer, or string. This tells the computer what type of information the variable holds. "Single" variables can hold numbers with decimal. "Integers" variables can hold whole numbers, while "String" variables holds text or a set of characters. If you don't dim a variable, the type would automatically be "Variant", which can hold almost all kinds of information. For example:

```
Option Explicit
Dim intNumber As Integer

intNumber = 31              ' This is ok
intNumber = "I didn't"      ' Error: type mismatch (intNumber is an integer
while "I didn't" is a string)
```

Dimming is especially important for arrays and matrices. For an array, next to the variable name, you enter the range of the array. For example:

```
Dim x(1 to 10) As Integer
```

Arrays will be covered more in depth later. Matrices are dimensioned almost exactly like arrays are, however, instead of the having only one dimension (1 to 20), matrices may have two: (1 to 20,1 to 5), or even three. Dimensioning can also be used to tell the computer that variables are public. This will be discussed later in the Scope section.

Note: If you don't dimension your variables, you might end up with many unexpected errors. It could be avoided by using the Option Explicit statement, which requires every variable to be defined; if not every variable used in the program is defined, VB raises an error: "Variable is not defined". To enable this, you just have to type Option Explicit at the very top of ALL your code in the current module. It's a very good practice to do so.

7 Chapter 25.10.2 on page 137

3.9 Simple output

The interaction between the user and the computer consists of both the input and output of data. The computer will not receive your commands if you don't have a mouse or keyboard which are used to input commands. And conversely, you wouldn't know what the computer is doing at all if there is no monitor or speaker which are used to output data. Therefore output is important.

3.9.1 Message boxes

One of the easiest form of output is message box. I'm sure you've seen a lot of message boxes in Windows. This is what the code of a normal message box should look like.

```
MsgBox("Hello world!")
```

Try it. Are you tired of the boring "Hello world!"? Let's make a fancier one:

```
MsgBox("Fatal error: Your computer will be shut down in five seconds.",
vbCritical, "System")
```

That will creep out quite a lot of people.

3.9.2 Printing

Note: The word "printing" here means using the Print statement, it's not about using the printer or printing files. Printing is a fairly simple part of Visual Basic, but also essential. Printing is used to output information to the user. It proves to be a valuable troubleshooting tool. Whenever printing, you need an object to print on, followed by of course, something to print. Printing may be used with various objects, however, the most common in the picture box. For the sake of simplicity, we are assuming you renamed the picture box as "pic". In this wikibook though, print is done mainly on picture boxes and forms:

```
pic.Print "Hello world!!"   'Prints message on picture box
Print "Hello world!!!"      'Prints message on current form
```

3.9.3 Spacing

There are various ways to alter how text is spaced when printing. The most common is the comma. A comma will go to the next print zone. Print zones are 15 characters long. You can think of it like pressing the tab key when typing something out. Remember that print zones are fixed, so if you've typed 1 letter, and then used a comma, then it will be a big space. If you type 13 characters and use a comma, it will not be a large space. For example:

```
Private Sub Form_Click()
    Me.Print "Hello", "Next Zone"
End Sub
```

Several new concepts are introduced in this example. The "Form_Click" contains a block of code and it is called to run when the user clicks on the current Form(Form1). 'Me' is the same as the current form (Form1). Don't be afraid to experiment. No matter what you do in VB, its always reversible. Now, the comma isn't all that versatile. Another feature is tab. Tab will move so many spaces from the BEGINNING of the line. Followed by tab in parentheses is the amount of characters spaces. For example:

```
Form1.Print "Hello"; Tab(10); "Yay"
```

This will NOT print "yay" 10 spaces after the O of "Hello". Rather it will print 10 spaces from the beginning of the line. You may use as many tabs as you want in the same print command. Although tab is useful, sometimes it is better to space things in relation to what has already been printed. This is where the space function comes in. The syntax of space is identical to that of tab. Space will move the next printed text so many spaces over from its CURRENT location. For example:

```
Pic.print "Hello"; Space(10); "Yay"
```

This will print the first Y of "Yay" 10 spaces to the right of the O in "Hello". It is important to note, if you write:

```
Pic.Print "Hello"
Pic.Print "Hello"
```

They will appear on separate lines as:

```
Hello
Hello
```

This can be easily dealt with in the need of having separate print statements print on the same line. You merely have to change the code to: (note the semicolon)

```
Pic.Print "Hello";
Pic.Print "Hello"
```

This will appear as:

```
HelloHello
```

If you want to make a blank line in between the two "hello"'s, then you may simply have a blank print statement WITHOUT a semicolon. For example:

```
Pic.Print "Hello"
Pic.Print
Pic.Print "Hello"
```

This will print as:

```
Hello

Hello
```

It is important to remember that if the first print has a semicolon at the end, often referred to as a trailing semicolon, the empty print will only reverse it, and print the second Hello on the next line, and no blank line will appear.

4 Simple Arithmetic

Visual Basic has all of the common arithmetical functions. It does not have complex numbers nor does it allow you to overload operators so manipulating complex numbers or matrices must be done by explicit function and subroutine calls.

4.1 Arithmetic Operators

The operators are infix and take two arguments: *arg1 operator arg2* except for unary plus and minus

4.1.1 Numeric Operators

Operator	**Comments**
+	Adds two numbers. Also concatenates strings but avoid this use because Visual Basic will try to convert the string to a number first and the results might not be what you expect.
-	Subtract the second number from the first.
- unary	negate the operand.
*	Multiply two numbers. The result will be promoted to whatever data type is needed to represent the size of the number if one of the numbers is already that type (see note below about odd behavior).
/	Normal division. Treats both operands as real numbers and returns a real result.
\	Integer division. Be careful with this because the arguments are converted to integers before the division is performed so use this with integer operands unless you comment your code carefully. Also note that "/" and "*" are evaluated before "\", so (1000 \ 13 * 3) is not equal to ((1000 \ 13) * 3)
Mod	Produces the remainder after integer division. Be careful with this as the interpretation of the modulus operator in mathematics is ambiguous. `a Mod b` gives the same answer as this expression: `a - (a \ b) * b`
^	Raises the first operand to the power of the second. In Visual Basic either or both operands may be negative. If all you want is a square or cube it is faster to explcitly multiply the number by itself the appropriate number of times.

For example:

```
Text2 = Text1 * 5
```

Will display the value in Text1, multiplied by 5, in Text2. E.g. if Text1 = 4 then Text2 will be equal to 20.

4.1.2 Order of operations

- Exponentiation (^)
- Multiplication and normal division (* and /)
- Integer division (\)
- Mod (Mod)
- Addition and subtraction (+,-)

General hint : If an expression uses more than (+,-,*) use all possible brackets to force the expression to be evaluated the way you are thinking it.

4.1.3 VB odd behavior note

VB considers "explicitly stated" integral numbers to be of type *Integer* (which must be between -32768 and 32767) if they are within (-32768, +32767) and gives an error if the result of arithmetic with them is more than 32768. This can be seen by trying

```
Debug.Print (17000 + 17000)
```

or

```
Debug.Print (17000 * 3)
```

which both cause an error. This can be solved (in a direct but ugly way) by enclosing numbers in *CLng()* (Convert to Long) so that

```
Debug.Print (CLng(17000) + CLng(17000))
```

or by using the type-declaration character *&* which specifies a Long constant:

```
Debug.Print (17000& + 17000&)
```

neither of which cause an error. To avoid having to think about this, avoid using explicit numbers in code and instead use "Long" variables and constants such as :

```
Const TEST_NUM As Long = 17000&
Debug.Print (TEST_NUM + TEST_NUM)

Dim TestNumVar As Long
TestNumVar = 17000
Debug.Print (TestNumVar + TestNumVar)
```

4.1.4 Boolean Arithmetic

Boolean operators use Boolean variables or integer variables where each individual bit is treated as a Boolean. There are six operators:

Operator:	**Meaning:**
Not	Negation
And	Conjuncton
Or	Disjunction (logical addition)
Xor	Exclusive Or
Eqv	Equivalence
Imp	Implication

When you construct logical expressions with these operators you get the following results:

A	B	A And B	A Or B	A Xor B	A Eqv B	A Imp B
T	T	T	T	F	T	T
T	F	F	T	T	F	F
F	T	F	T	T	F	T
F	F	F	F	F	T	T

4.2 Comparison Operators

These operators, composed of <, > and =, are use to decide whether one value is smaller than, larger than, or equal to another.

For example:

```
Dim i
i = 50
If i < 0 Then
    MsgBox "i is less than 0"
ElseIf i <= 100 And i >= 0 Then
    MsgBox "i is less than or equal to one hundred and greater than or equal
to 0"
ElseIf i > 100 And i < 200 Then
    MsgBox "i is greater than one hundred less than 200"
Else
    MsgBox "i is greater than or equal to 200"
End if
```

Caution! Due to the internal structure of floating-point numbers (Single and Double), do not use = or <> to compare them. Instead, use a small value (usually called Epsilon) as a "maximum difference". For example:

```
' This returns False :
Debug.Print (Sqr(1.234) * Sqr(1.234)) = 1.234
' This returns True :
E = 0.000001
Debug.Print Abs((Sqr(1.234) * Sqr(1.234)) - 1.234) < E
```

Operator	Meaning
=	Equality
<>	Inequality
<	Less than
>	Greater than
>=	Greater than or equal to. Or put another way: *not less than*
<=	Less than or equal to. Or put another way: *not greater than*

4.3 Built in Arithmetic Functions

There are not many native mathematical functions in Visual basic but this doesn't mean that you can't do significant calculations with it.

Abs(x)

returns the absolute value of x, that is, it removes a minus sign if there is one. Examples: Abs(3)=3 ; Abs(-3)=3

Exp(x)

returns the value e^x. e is Euler's constant, the base of natural logarithms.

Log(x)

the *Neperian ('Natural', e base) logarithm* of *x*.

Randomize(x)

not really a mathematical function because it is actually a subroutine. This initializes the random number generator.

Rnd(x)

produces the next random number in the series. Please read that sentence again! the random numbers aren't really random, they are instead pseudo-random. If you initialize the random number generator with the same number each time you start a program then you will get the same series of values from *Rnd()*

Round(x,n)

returns a real number rounded to n decimal places (uses Banker's rounding).

Sgn(x)

returns plus one if x is positive, minus one if it is negative, zero if x is identically zero. Sgn(-5)=-1 ; Sgn(5)=1 ; Sgn(0)=0

Sqr(x)

square root of x. Example: Sqr(25)=5. x must be non-negative. Thus Sqr(-25) will generate an error

4.3.1 Derived Functions

If you want logarithms to some other base you can use this expression:

```
Log(x, base) = Log(x) / Log(base)
```

And to calculate the n^{th} root of a number (cube root, ...)

```
RootN(x, n) = x ^ (1.0 / n)
```

4.4 Trigonometrical Functions

Visual Basic has the usual simple trigonometric functions, sin, cos, tan, but if you want some of the more unusual ones or inverses you will need to write some simple functions.

Remember that the angles must be supplied as *radians*

```
radians = degrees * &pi; / 180

ArcSin(x) = Atn(x / Sqr(-x * x + 1))

ArcCos(x) = Atn(-x / Sqr(-x * x + 1)) + 2 * Atn(1)
```

Notice that the range of applicability of these expressions is limited to the range -1<=x<=1.

Here are some more:

Secant	Sec(x) = 1 / Cos(x)

Cosecant	Cosec(x) = 1 / Sin(x)
Cotangent	Cotan(x) = 1 / Tan(x)
Inverse Sine	Arcsin(x) = Atn(x / Sqr(-x * x + 1))
Inverse Cosine	Arccos(x) = Atn(-x / Sqr(-x * x + 1)) + 2 * Atn(1)
Inverse Secant	Arcsec(x) = Atn(x / Sqr(x * x - 1)) + Sgn((x) - 1) * (2 * Atn(1))
Inverse Cosecant	Arccosec(x) = Atn(x / Sqr(x * x - 1)) + (Sgn(x) - 1) * (2 * Atn(1))
Inverse Cotangent	Arccotan(x) = -Atn(x) + 2 * Atn(1)
Hyperbolic Sine	HSin(x) = (Exp(x) - Exp(-x)) / 2
Hyperbolic Cosine	HCos(x) = (Exp(x) + Exp(-x)) / 2
Hyperbolic Tangent	HTan(x) = (Exp(x) - Exp(-x)) / (Exp(x) + Exp(-x))
Hyperbolic Secant	HSec(x) = 2 / (Exp(x) + Exp(-x))
Hyperbolic Cosecant	HCosec(x) = 2 / (Exp(x) - Exp(-x))
Hyperbolic Cotangent	HCotan(x) = (Exp(x) + Exp(-x)) / (Exp(x) - Exp(-x))
Inverse Hyperbolic Sine	HArcsin(x) = Log(x + Sqr(x * x + 1))
Inverse Hyperbolic Cosine	HArccos(x) = Log(x + Sqr(x * x - 1))
Inverse Hyperbolic Tangent	HArctan(x) = Log((1 + x) / (1 - x)) / 2
Inverse Hyperbolic Secant	HArcsec(x) = Log((Sqr(-x * x + 1) + 1) / x)
Inverse Hyperbolic Cosecant	HArccosec(x) = Log((Sgn(x) * Sqr(x * x + 1) + 1) / x)
Inverse Hyperbolic Cotangent	HArccotan(x) = Log((x + 1) / (x - 1)) / 2

The very useful atan2 function (calculate the angle in all four quadrants of a vector) can be simulated like this:

```
Public Const Pi As Double = 3.14159265358979

 Public Function Atan2(ByVal y As Double, ByVal x As Double) As Double

   If y > 0 Then
     If x >= y Then
       Atan2 = Atn(y / x)
     ElseIf x <= -y Then
       Atan2 = Atn(y / x) + Pi
     Else
       Atan2 = Pi / 2 - Atn(x / y)
     End If
   Else
     If x >= -y Then
       Atan2 = Atn(y / x)
     ElseIf x <= y Then
       Atan2 = Atn(y / x) - Pi
     Else
       Atan2 = -Atn(x / y) - Pi / 2
     End If
   End If

 End Function
```

Other functions:

```
ASin(x) = Atan2(x, Sqr(1 - x * x))
ACos(x) = Atan2(Sqr(1 - x * x), x)
```

5 Branching

A branch is a point at which your program must make a choice. With these data structures, it is now possible to make programs that can have multiple outcomes. You may be familiar with the first type from Algebra[1], an If-Then statement, or *If P then Q*. And they work pretty much the same. Another type is the *Select Case*, this may prove to be easier at times.

Another name for this concept is *conditional clauses*. Conditional clauses are blocks of code that will only execute if a particular expression (the condition) is true.

5.1 If...Then Statement

If...Then statements are some of the most basic statements in all of programming. Every language has them, in some form or another. In Visual Basic, the syntax for an If...Then statement is as follows:

```
If (condition) Then
    (reaction)
End If
```

- **condition** - a set of test(s) that the program executes.
- **reaction** - the instructions that the program follows when the condition returns true. The condition returns true if it passes the test and returns false if it fails the test.

The condition can be anything from

```
If X = 1 Then
    MsgBox "X = 1"
End If
```

to

```
If InStr(1, sName, " ") > 0 Then
    sName = """" & sName & """"
End If
```

If there is only one reaction to the condition, the statement can also be expressed without the End If:

```
If Y + 3 = 7 Then MsgBox "Y + 3 DOES = 7"
```

1 http://en.wikibooks.org/wiki/Algebra

There are also other parts to these statements to make them more complex. Two other terms that can be used are Else, and ElseIf.

Else will, if the condition is false, do whatever comes between the Else statement and the End If statement.

ElseIf will, if the condition **directly proceeding it** is false, check for another condition and go from there.

5.2 If..Then..Else Statement

The If..Then..Else statement is the simplest of the conditional statements. They are also called branches, as when the program arrives at an "If" statement during its execution, control will "branch" off into one of two or more "directions". An If-Else statement is generally in the following form:

```
If condition Then
    statement
Else
    other statement
End If
```

If the original condition is met, then all the code within the first statement is executed. The optional Else section specifies an alternative statement that will be executed if the condition is false. The If-Else statement can be extended to the following form:

```
If condition Then
    statement
ElseIf condition Then
    other statement
ElseIf condition Then
    other statement
    ...
Else
    another statement
End If
```

Only one statement in the entire block will be executed. This statement will be the first one with a condition which evaluates to be true. The concept of an If-Else-If structure is easier to understand with the aid of an example:

```
Dim Temperature As Double
...
If Temperature >= 40.0 Then
    Debug.Print "It's extremely hot"
ElseIf 30.0 <= Temperature And Temperature<=39.0 Then
    Debug.Print "It's hot"
ElseIf 20.0 <= Temperature And Temperature <= 29.0 Then
    Debug.Print "It's warm"
ElseIf 10.0 <= Temperature And temperature <= 19.0 Then
    Debug.Print "It's cool"
ElseIf 0.0 <= Temperature And Temperature <= 9.0 Then
    Debug.Print "It's cold"
Else
    Debug.Print "It's freezing"
End If
```

5.2.1 Optimizing hints

When this program executes, the computer will check all conditions in order until one of them matches its concept of truth. As soon as this occurs, the program will execute the statement immediately following the condition and continue on, without checking any other condition for truth. For this reason, when you are trying to optimize a program, it is a good idea to sort your If..Then..Else conditions in order of descending likelihood. This will ensure that in the most common scenarios, the computer has to do less work, as it will most likely only have to check one or two *branches* before it finds the statement which it should execute. However, when writing programs for the first time, try not to think about this too much lest you find yourself undertaking premature optimization.

In Visual Basic Classic conditional statements with more than one conditional do not use short-circuit evaluation. In order to mimic C/C++'s short-circuit evaluation, use ElseIf as described in the example above. In fact for complicated expressions explicit *If..Then..ElseIf* statements are clearer and easier to read than the equivalent short circuit expression.

5.3 Select Case

Often it is necessary to compare one specific variable against several constant expressions. For this kind of conditional expression the Select Case is used. The above example is such a case and could also be written like this:

```
Select Case Temperature
    Case Is >= 40#
        Debug.Print "It's extremely hot"
    Case 30# To 39#
        Debug.Print "It's hot"
    Case 20# To 29#
        Debug.Print "It's warm"
    Case 10# To 19#
        Debug.Print "It's cool"
    Case 0# To 9#
        Debug.Print "It's cold"
    Case Else
        Debug.Print "It's freezing"
End Select
```

5.4 Unconditionals

Unconditionals let you change the flow of your program without a condition. You should be careful when using unconditionals. Often they make programs difficult to understand. Read *Isn't goto evil?* below for more information.

5.4.1 Exit

End a function, subroutine or property and return to the calling procedure or function. Note that in Visual Basic returning from a function and assigning a return value requires two separate statements.

For procedures:

```
Exit Sub
```

For functions:

```
Exit function
```

For properties:

```
Exit Property
```

5.4.2 End

This simple command ends the execution of the program

For example:

```
Private Sub cmd1_Click()
    End
End Sub
```

In this example, when the button cmd1 is clicked, the program will terminate. The **End** command is provided for backward compatibility and is rarely (if ever) needed to end a VB program. The proper way to end a VB program is to release all object references, stop any running timers, and exit out of any executing procedures. In a small program this could mean to simply unload all the forms. When there are no active object references and no code running, VB will terminate the program in a graceful manner. Using **End** to end the program will terminate the program in an ungraceful manner (some things may not get shut down properly).

5.4.3 Goto

Transfer control to the statement after the label (or line number).

```
    Goto Label
    Dont_Do_Something
Label:
    ...
```

In Visual Basic the target of a Goto statement must be in the same procedure; this prevents abuse of the feature by making it impossible to jump into the middle of another subroutine.

Isn't Goto Evil?

One often hears that Goto is evil and one should avoid using goto. But it is often overlooked that any exit statement which is not the last statement inside a procedure or function is also an unconditional statement - a goto in disguise.

Therefore if you have functions and procedures with more than one exit statement you can just as well use goto. When it comes down to readability the following two samples are almost the same:

```
Sub Use_Exit
    Do_Something
    If Test Then
        Exit Sub
    End If
    Do_Something_Else
End Sub

Sub Use_Goto is
    Do_Something
    If Test Then
        Goto Exit_Use_Goto
    End If
    Do_Something_Else
Exit_Use_Goto:
End Sub
```

In the pure structured approach, neither goto nor multiple exit statements are needed:

```
Sub Use_If is
    Do_Something
    If Not Test Then
        Do_Something_Else
    End If
End Sub
```

A couple of notes:

- Return is a reserved keyword and can only be used with a matching Gosub and that the Gosub must reside in the same function, sub or property. It cannot be used to return from a function, sub or property - maybe it should read Exit Sub.
- Error handling in Visual Basic does need the use of Goto, but with a different syntax.
- Visual Basic, believe it or not, supports line numbering, so for example Goto 20 is perfectly acceptable if line 20 exists!
- Using Exit instead of Goto as shown above has some side effects. Calling Exit Sub clears a current error state, i.e. it is the equivalent of an Err.Clear.

Compatibility with VB.Net

If you are concerned that you might want to port your code to VB.Net without recasting it later you should probably avoid *goto* altogether.

One alternative is the pseudo loop. Its purpose is to fake a *goto* target that occurs later in the routine:

```
Sub Use_PseudoLoop

    Do 'Start the pseudo loop.  Exists only so that you can use Exit Do
        Try_Something
        If Test Then
            Exit Do
        End If
        Try_Again
        If Test Then
```

```
            Exit Do
        End If
        Try_Try_Again
    Loop While False ' will never loop
    Do_One_Last_Thing

End Sub
```

Of course this example can be recast using *If..Then..Else..End If* without the need for any Exit statements at all so it is a little artificial:

```
Sub Use_IfThen

    Try_Something
    If Not Test Then
        Try_Again
        If Not Test Then
            Try_Try_Again
        End If
    End If
    Do_One_Last_Thing

End Sub
```

6 Loops

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Visual Basic has three main types of loops: *for..next* loops, *do* loops and *while* loops.

Note: 'Debug' may be a reserved word in Visual Basic, and this may cause the code samples shown here to fail for some versions of Visual Basic.

6.1 For..Next Loops

The syntax of a *For..Next* loop has three components: a *counter*, a *range*, and a *step*. A basic *for..next* loop appears as follows:

```
For X = 1 To 100 Step 2
 Debug.Print X
Next X
```

In this example, X is the counter, "1 to 100" is the range, and "2" is the step.

The variable reference in the *Next* part of the statement is optional and it is common practice to leave this out. There is no ambiguity in doing this if code is correctly indented.

When a *For..Next* loop is initialized, the counter is set to the first number in the range; in this case, X is set to 1. The program then executes any code between the *for* and *next* statements normally. Upon reaching the *next* statement, the program returns to the *for* statement and increases the value of the counter by the step. In this instance, *X* will be increased to *3* on the second iteration, *5* on the third, etc.

To change the amount by which the *counter* variable increases on each *iteration*, simply change the value of the *step*. For example, if you use a *step 3*, *X* will increase from *1* to *4*, then to *7, 10, 13*, and so on. When the step is not explicitly stated, *1* is used by default. (Note that the *step* can be a negative value. For instance, `for X = 100 to 1 step -1` would decrease the value of X from 100 to 99 to 98, etc.)

When X reaches the end of the range in the range (100 in the example above), the loop will cease to execute, and the program will continue to the code beyond the *next* statement.

It is possible to edit the value of the counter variable within a *for..next* loop, although this is generally considered bad programming practice:

```
For X = 1 To 100 Step 1
 Debug.Print X
 X = 7
Next
```

While you may on rare occasions find good reasons to edit the counter in this manner, the example above illustrates one potential pitfall:

Because X is set to 7 at the end of every iteration, this code creates an infinite loop. To avoid this and other unexpected behavior, use extreme caution when editing the counter variable!

It is not required by the compiler that you specify the name of the loop variable in the *Next* statement but it will be checked by the compiler if you do, so it is a small help in writing correct programs.

6.1.1 For loop on list

Another very common situation is the need for a loop which *enumerates* every element of a list. The following sample code shows you how to do this:

```
Dim v As Variant

For Each v In list
 Debug.Print v
Next
```

The list is commonly a Collection or Array, but can be any other object that implements an *enumerator*. Note that the iterating variable has to be either a **Variant**, **Object** or class that matches the type of elements in the list.

6.2 Do Loops

Do loops are a bit more flexible than For loops, but should generally only be used when necessary. Do loops come in the following formats:

- Do while
- Do until
- Loop while
- Loop until

While loops (both do while and loop while) will continue to execute as long as a certain conditional is true. An Until loop will loop as long as a certain condition is false, on the other hand. The only difference between putting either While or Until in the Do section or the Loop section, is that Do checks when the loop starts, and Loop checks when the loop ends. An example of a basic loop is as follows:

```
Do
 Debug.Print "hello"
 x = x + 1
Loop Until x = 10
```

This loop will print hello several times, depending on the initial value of x. As you may have noticed, Do loops have no built in counters. However, they may be made manually as shown above. In this case, I chose x as my counter variable, and every time the loop execute, x increase itself by one. When X reaches 10, the loop will cease to execute. The

advantage of Do loops is that you may exit at any time whenever any certain conditional is met. You may have it loop as long as a certain variable is false, or true, or as long as a variable remains in a certain range.

6.2.1 Endless loop: *Do..Loop*

The endless loop is a loop which never ends and the statements inside are repeated forever. Never is meant as a relative term here - if the computer is switched off then even endless loops will end very abruptly.

```
Do
 Do_Something
Loop
```

In Visual Basic you cannot label the loop but you can of course place a label just before it, inside it or just after it if you wish.

6.2.2 Loop with condition at the beginning: *Do While..Loop*

This loop has a condition at the beginning. The statements are repeated as long as the condition is met. If the condition is not met at the very beginning then the statements inside the loop are never executed.

```
Do While X <= 5
 X = Calculate_Something
Loop
```

6.2.3 Loop with condition at the end: *Do..Loop Until*

This loop has a condition at the end and the statements are repeated until the condition is met. Since the check is at the end the statements are at least executed once.

```
Do
 X = Calculate_Something
Loop Until X > 5
```

6.2.4 Loop with condition in the middle:*Do..Exit Do..Loop*

Sometimes you need to first make a calculation and exit the loop when a certain criterion is met. However when the criterion is not met there is something else to be done. Hence you need a loop where the exit condition is in the middle.

```
Do
 X = Calculate_Something
 If X > 10 then
  Exit Do
 End If
 Do_Something (X)
Loop
```

In Visual Basic you can also have more than one exit statement. You cannot exit named outer loops using *Exit Do* because Visual Basic does not provide named loops; you can of course use Goto instead to jump to a label that follows the outer loop.

6.3 While Loops

While loops are similar to Do loops except that the tested condition always appears at the top of the loop. If on the first entry into the loop block the condition is false, the contents of the loop are never executed. The condition is retested before every loop iteration.

An example of a While loop is as follows:

```
price = 2

While price < 64
    Debug.Print "Price = " & price
    price = price ^ 2
Wend

Debug.Print "Price = " & price & ":  Too much for the market to bear!"
```

The While loop will run until the condition tests false - or until an "Exit While" statement is encountered.

6.4 Nested Loops

A nested loop is any type of loop inside an already existing loop. They can involve any type of loop. For this, we will use For loops. It is important to remember that the inner loop will execute its normal amount multiplied by how many times the outer loop runs. For example:

```
For i = 1 To 10
 For j = 1 To 2
  Debug.Print "hi"
 Next
Next
```

This will print the word 'hi' twenty times. Upon the first pass of the i loop, it will run the j loop twice. Then on the second pass of the i loop, it will run the j loop another two times, and so on.

7 Strings

Visual Basic has a traditional set of built in string operations. Unlike many languages Visual Basic strings are always Unicode[1] so they can hold any character. They are also always dynamically allocated and are of almost unlimited length (theoretically up to about 2^31 characters, about 2000 million).

Note that Unicode uses more than one byte to represent each character. As far as VB is concerned Unicode characters are two bytes which gives 2^16 or 65536 possible values. This is enough even for Chinese, Japanese and Korean character sets (*CJK*). In fact Unicode defines 17 planes each of which has room for 2^16 code points but VB (and Windows) uses only the Basic Multilingual Plane[2] (*BMP*).

See What Are VB Strings?[3] for a concise explanation of the internal workings of Visual Basic Classic strings.

7.1 Built In String Functions

Visual Basic provides a reasonable set of traditional functions for manipulating and searching strings. These functions are usually all that is needed for most programs that are not primarily concerned with text processing.

7.2 Regular Expressions

A regular expression is a character string in which some characters have special meanings. Such a string can be used to search for substrings in another string in much more sophisticated ways than the built in InStr function.

For example this expression:

```
"(dog|cat)"
```

will match either 'dog' or 'cat'.

Visual Basic has no built in regular expression functions, but these are available in the VBScript regular expression library. If your program does a lot of text processing *regular*

1 http://en.wikipedia.org/wiki/Unicode
2 http://en.wikipedia.org/wiki/Basic%20Multilingual%20Plane
3 http://web.archive.org/web/20080626021549/http://www.romanpress.com/Articles/Strings_R/Strings.htm

expressions are definitely worth learning even though they may seem intimidating at the start. In practice most programmers find that the more arcane expressions are rarely used and the same idioms are recycled over and over so there is really not as much to learn as it at first appears.

8 Built In String Functions

8.1 Comparison

Two strings are **equal by value** if they have the same content:

- If "Hello" = "Hello" Then MsgBox "A"

The statement **Option Compare Text** can be placed at the top of a module to make the comparison **case-insensitive**, impacting =, <, >, <=, >=, <>:

```
Option Compare Text
Sub Test()
  If "Hello" = "hello" Then MsgBox "A"
  If "aaa" < "BBB" Then MsgBox "B"
End Sub
```

To test whether two strings are **equal by reference**, that is, whether they start at the same address, you can use StrPtr function.

To test whether a string is **greater than or less** than another using lexicographical order, you can use <, >, <=, >=, <>. To decide whether a string is less than another, the two are compared character by character and as soon as a character from one string is found to have a lower ASCII code than the corresponding (same position) character in the other string, the first is declared to be *less than* the second. The converse test also works.

Another way of testing whether strings are equal or greater than one another is the **StrComp** function. Unlike =, <, >, etc., the StrComp function has an optional argument that controls whether the comparison is case-sensitive.

Example:

```
strHello = "hello": strHello2 = "Hello"
If StrComp(strHello, strHello2, vbTextCompare) = 0 Then
  Debug.Print "The strings are the same."
End If
```

Links:

- StrComp Function[1], Access 2007, office.microsoft.com
- StrComp Function[2], Office 2013, msdn.microsoft.com

1 http://office.microsoft.com/en-001/access-help/strcomp-function-HA001228914.aspx
2 http://msdn.microsoft.com/en-us/library/office/gg264346.aspx

8.2 Concatenation

The operator intended to perform string contatenation is *&*. The operator *+* can sometimes be used to the same effect, but not always:

```
a = "123"
b = "456"
c = a & b 'Yields 123456
d = a + b 'Yields 123456: applied to two strings
Debug.Print c, d

a = 123   'A number, not a string
b = "456" 'A string
c = a & b 'Yields 123456
d = a + b 'Yields 579: applied to a number and to a string
Debug.Print c, d
```

8.3 Containment

To find out if one string is a substring of another, use the InStr function as follows:

```
If InStr("Hello","He") > 0 Then
  MsgBox "The second string is contained in the first string."
End If
If InStr("He","Hello") > 0 Then
  MsgBox "This never gets printed; wrong argument order."
End If
```

InStr function returns the position of the substring if it is found or zero otherwise.

The two-argument use of InStr function is case-sensitive. A case-insensitive containment test can be done as follows:

```
If InStr(UCase("Hello"), UCase("he")) > 0 Then
  MsgBox "The second string is contained in the first string when we " & _
         "disregard the case."
End If
```

8.4 Replacing

To replace a string with another string inside a third string, use the built-in function Replace as follows:

```
Result = Replace("Teh Beatles", "Teh", "The") 'results into "The Beatles"
```

8.5 Indexing and Substrings

Strings can be used almost as if they were lists of characters. The **nth character in a string** can be returned by subscripting:

```
Mid$(String1, n, 1)
```

The values of n start from 1 rather than from 0.

It is also possible to return a **substring** of a string. The same function is used but instead of specifying 1, you specify the length of the substring:

```
Offset = 2: Length = 3
Debug.Print Mid$("abcdef", Offset , Length) 'Prints bcd
```

If you ask for more characters than are available, you get just what there is, no error is raised:

```
Debug.Print Mid$("abcdef", 2, 33) 'Prints bcdef
```

You can also use **Mid$** on the **left-hand side** of an assignment:

```
String1 = "aaaaaa"
Debug.Print String1 'Prints aaaaaa
Mid$(String1, 2, 3) = "bbb"
Debug.Print String1 'Prints abbbaa
Mid$(String1, 3, 1) = "cccccc"
Debug.Print String1 'Prints abcbaa
Mid$(String1, 2, 3) = "d"
Debug.Print String1 'Prints adcbaa
```

When Mid$ is on the left, it doesn't change the total length of the string: it just replaces the specified number of characters. If the right-hand side specifies fewer characters than are asked for, only that number of characters is replaced; if it specifies more, only the number asked for is used.

Links:

- MID, MIDB[3], Excel 2003, office.microsoft.com

8.6 String constants

String constants can be declared like any other constant:

```
Const s As String = "abcdef"
```

8.7 String Functions

Strings are not objects so they do not have methods but there is a number of functions that manipulate strings. Note that none of the functions modify the original string, except for Mid$ when it is on the left hand side of an assignment statement:

Asc

3 http://office.microsoft.com/en-gb/excel-help/mid-midb-HP005209175.aspx

Returns the integer code of the first character of the string. The inverse function would be *Chr*.

Len

Returns the length of the string.

InStr

Returns the character index of the first occurrence of the substring in a string or zero if the substring is not found.

InstrB

Like InStr except that it returns the byte position. It has to be remembered, that Visual Basic 6 strings are Unicode strings.

InstrRev

Like InStr except that it returns the character position of the last occurrence of the substring.

Left$

Returns the specified number of characters from the beginning of the string. If there are fewer characters in the string Left$ returns the whole string, no error is raised,

Mid$

Returns a number of characters starting at the given position, on the left hand side it replaces those characters,

Right$

Returns the specified number of characters from the end of the string, if there are not that many characters, then Right$ returns the whole string.

IsNumeric

Returns true if the string looks like a number.

LTrim$, RTrim$, Trim$

Returns a copy of the string with leading, trailing or leading and trailing spaces removed respectively. Note that only ASCII spaces (character code 32) are removed, other whitespace characters such as tabs are treated as non-spaces.

LCase$, UCase

Converts the whole string to lower case or upper case respectively.

Val

Returns a number corresponding to the number found at the start of the string. Note that *Val* is not locale aware, which means that it always expects decimal points regardless of the regional settings of your computer; if you are reading from a comma delimited file this is probably the function you want to use.

Str

Returns a string corresponding to the given number. Like *Val* this is not locale aware. This is the function you should use if you are creating a text file containing numbers to be read on someone else's computer.

CStr

Converts the expression to a string. This procedure *is* locale aware and the correct function to use if converting numbers and differently typed values to strings for user-display. Usually it is unnecessary because Visual Basic automatically converts when necessary and uses *Regional Settings* to do so.

Format$

Converts a number to a string using a specific format. The format is provided as a string of characters, that shows how many digits should be given before and after the decimal point. Like *CStr*, *Format$* is locale aware so the decimal separator will be whatever is specified in the user's *Regional Settings*. *Format$* also provides for conversion of dates to various built-in and custom string formats.

CBool, CByte, CCur, CInt, CLng, CSng, CDbl, CDec

Locale aware conversions to *Boolean*, *Byte*, *Currency*, *Integer*, *Long*, *Single*, *Double*, *Decimal*.

Split

Chops a string into pieces and returns a *Variant Array*. If no delimiter is specified then spaces will be used. Delimiters may be any string of any length. Two adjacent delimiters delimit an empty string.

Hex$

Returns a string of *Hex* characters representing a number.

Oct$

Returns a string of *Octal* characters representing a number.

Replace$

Returns a string with occurrences of a specified substring replaced with a new string. Note that the substring and the new string do not have to be the same size.

StrComp

Returns -1 if the first string is less than the second, 0 if they are identical, +1 if the first is greater than the second. Takes an optional argument that determines the comparison algorithm: vbBinary for exact comparisons using the character codes, vbTextCompare for case insensitive comparisons.

8.8 Quotes in strings

Because the double quote (") is used to delimit strings, you can't use it directly to specify a quote within a string. For example

```
' Does not work - syntax error
Debug.Print "Fred says "Hi" to you"
```

One way is to use that Chr() function to specify the character code (34)

```
' Works fine
Debug.Print "Fred says " & Chr$(34) & "Hi" & Chr$(34) & " to you"
```

Another is to double the double-quotes. You might find this more or less readable than the above way.

```
' Works fine too
Debug.Print "Fred says ""Hi"" to you"
```

8.9 Startswith and Endswith

Visual Basic does not have functions "startsWith" (or "BeginsWith") and "endsWith" found in some other programming languages. But it has "Like" comparison operator used for simple pattern matching that does the job when used with "*" to stand for "any string of characters":

```
If "Hello World" Like "Hello*" Then MsgBox "It starts."
If "Hello World" Like "*World" Then MsgBox "It ends."
If LCase("Hello World") Like "hello*" Then MsgBox "It starts, case insensitive."
If LCase("Hello World") Like "*world" Then MsgBox "It ends, case insensitive."
Ending = "World"
If "Hello World" Like "*" & Ending Then MsgBox "It ends."
```

Furthermore, you can write these functions yourself using "Left" function:

```
Function StartsWith(Str1 As String, Str2 As String, Optional CaseIns As Boolean)
 As Boolean
  StartsWith = False
  If CaseIns Then 'Case insensitive
    If Left(LCase(Str1), Len(Str2)) = LCase(Str2) Then
      StartsWith = True
    End If
  Else
    If Left(Str1, Len(Str2)) = Str2 Then
      StartsWith = True
    End If
  End If
End Function

Function EndsWith(Str1 As String, Str2 As String, Optional CaseIns As Boolean)
 As Boolean
```

```
  EndsWith = False
  If CaseIns Then 'Case insensitive
    If Right(LCase(Str1), Len(Str2)) = LCase(Str2) Then
      EndsWith = True
    End If
  Else
    If Right(Str1, Len(Str2)) = Str2 Then
      EndsWith = True
    End If
  End If
End Function
```

For a one-off comparison to a constant string, a function call may be an overkill:

```
If Left(String1, 9) = "Beginning" Then
  'Starts with, case-sensitive
  ...
Else
  ...
End If

If Right(String1, 3) = "end" Then
  'Ends with, case-sensitive
  ...
Else
  ...
End If
```

8.10 Pattern Matching

You can do simple pattern matching with **Like** keyword; for complex pattern matching, see ../Regular Expressions/[4]. The special characters in the patterns of *Like* include ? for a single char, * for any number of chars, # for a single decimal digit, [...] for a single char in the list, and [!...] for a single char not in the list.

Examples:

```
If "Hello World" Like "Hello*" Then MsgBox "A"
If "Hello World" Like "*World" Then MsgBox "B"
If "Itam" Like "It?m" Then MsgBox "G"
If "AB345" Like "[A-Z][A-Z]###" Then MsgBox "C"
If "Ab345" Like "[a-zA-Z][a-zA-Z]###" Then MsgBox "D"
If "Item" Like "[!;][!;][!;][!;]" Then MsgBox "E"
If Not "It;m" Like "[!;][!;][!;][!;]" Then MsgBox "F"
```

Links:

- Like Operator[5], Visual Basic for Applications, msdn.microsoft.com
- Like Operator[6], Visual Studio 2005, msdn.microsoft.com

4 Chapter 8.10 on page 51
5 http://msdn.microsoft.com/en-us/library/office/gg251796.aspx
6 http://msdn.microsoft.com/en-us/library/swf8kaxw%28v=vs.80%29.aspx

9 Regular Expressions

Sometimes, the built in string functions are not the most convenient or elegant solution to the problem at hand. If the task involves manipulating complicated patterns of characters, *regular expressions* can be a more effective tool than sequences of simple string functions.

Visual Basic has no built-in support for regular expressions. It can use regular expressions via VBScript Regular Expression Library, though. If you have Internet Explorer installed, you almost certainly have the library. To use it, you must add a reference to the project; on the *Project* menu choose *References* and scroll down to *Microsoft VBScript Regular Expressions*. There might be more than one version; if so, choose the one with the highest version number, unless you have some particular reason to choose an old version, such as compatibility with that version on another machine.

9.1 Class outline

Class outline of VBScript.RegExp class:

- Attributes
  - RegExp.Pattern
  - RegExp.Global
  - RegExp.IgnoreCase
  - RegExp.MultiLine
- Methods
  - RegExp.Test
  - RegExp.Replace
  - RegExp.Execute

9.2 Constructing a regexp

A method of constructing a regular expression object:

```
Set Regexp = CreateObject("VBScript.RegExp")
Regexp.Pattern = "[0-9][0-9]*"
```

A method of constructing a regular expression object that requires that, in Excel, you set a reference to Microsoft VBScript Regular Expressions:

```
Set Regexp = new RegExp
Regexp.Pattern = "[0-9][0-9]*"
```

9.3 Testing for match

An example of testing for match of a regular expression

```
  Set RegExp = CreateObject("VBScript.RegExp")
  RegExp.Pattern = "[0-9][0-9]*"
  If RegExp.Test("354647") Then
    MsgBox "Test 1 passed."
  End If
  If RegExp.Test("a354647") Then
    MsgBox "Test 2 passed." 'This one passes, as the matching is not a
whole-string one
  End If
  If RegExp.Test("abc") Then
    MsgBox "Test 3 passed." 'This one does not pass
  End If
```

An example of testing for match in which the whole string has to match:

```
  Set RegExp = CreateObject("VBScript.RegExp")
  RegExp.Pattern = "^[0-9][0-9]*$"
  If RegExp.Test("354647") Then
    MsgBox "Test 1 passed."
  End If
  If RegExp.Test("a354647") Then
    MsgBox "Test 2 passed." 'This one does not pass
  End If
```

9.4 Finding matches

An example of iterating through the collection of all the matches of a regular expression in a string:

```
  Set Regexp = CreateObject("VBScript.RegExp")
  Regexp.Pattern = "a.*?z"
  Regexp.Global = True 'Without global, only the first match is found
  Set Matches = Regex.Execute("aaz abz acz ad1z")
  For Each Match In Matches
    MsgBox "A match: " & Match
  Next
```

9.5 Finding groups

An example of accessing matched groups:

```
  Set Regexp = CreateObject("VBScript.RegExp")
  Regexp.Pattern = "(a*) *(b*)"
  Regexp.Global = True
  Set Matches = Regexp.Execute("aaa bbb")
  For Each Match In Matches
    FirstGroup = Match.SubMatches(0) '=aaa
    SecondGroup = Match.SubMatches(1) '=bbb
  Next
```

9.6 Replacing

An example of replacing all sequences of dashes with a single dash:

```
Set Regexp = CreateObject("VBScript.RegExp")
Regexp.Pattern = "--*"
Regexp.Global = True
Result = Regexp.Replace("A-B--C----D", "-") '="A-B-C-D"
```

An example of replacing doubled strings with their single version with the use of two sorts of *backreference*:

```
Set Regexp = CreateObject("VBScript.RegExp")
Regexp.Pattern = "(.*)\1"
Regexp.Global = True
Result = Regexp.Replace("hellohello", "$1") '="hello"
```

9.7 Splitting

There is no direct support for splitting by a regular expression, but there is a workaround. If you can assume that the split string does not contain Chr(1), you can first replace the separator regular expression with Chr(1), and then use the non-regexp split function on Chr(1).

An example of splitting by a non-zero number of spaces:

```
SplitString = "a b  c   d"
Set Regexp = CreateObject("VBScript.RegExp")
Regexp.Pattern = "  *"
Regexp.Global = True
Result = Regexp.Replace(SplitString , Chr(1))
SplitArray = Split(Result, Chr(1))
For Each Element In SplitArray
  MsgBox Element
Next
```

9.8 Example application

For many beginning programmers, the ideas behind regular expressions are so foreign that it might be worth presenting a simple example before discussing the theory. The example given is in fact the beginning of an application for scraping web pages to retrieve source code so it is relevant too.

Imagine that you need to parse a web page to pick up the major headings and the content to which the headings refer. Such a web page might look like this:

Figure 2 frame

```
<nowiki>
  <html>
    <head>
```

```
<nowiki>
  <html>
    <head>
      <title>RegEx Example</title>
    </head>
    <body>
      <h1>RegEx Example</h1>
        aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
        <h2>Level Two in RegEx Example</h2>
          bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
      <h1>Level One</h1>
        ccccccccccccccccccccccccccccccccccc
        <h2>Level Two in Level One</h2>
          ddddddddddddddddddddddddddddddddddddd
    </body>
  </html>
</nowiki>
```

What we want to do is extract the text in the two *h1 elements* and all the text between the first *h1* and the second *h1* as well as all the text between the second *h1 element* and the end of body tag.

We could store the results in an array that looks like this:

"RegEx Example"	" aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\n Level Two in RegEx Example \nbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb"
"Level One"	" ccccccccccccccccccccccccccccccccccc-ccc\n Level Two in Level One \n ddddddddddddddddddddddddddddddddddddd"

The *\n* character sequences represent end of line marks. These could be any of *carriage return*, *line feed* or *carriage return* followed by *line feed*.

A regular expression specifies patterns of characters to be matched and the result of the matching process is a list of sub-strings that match either the whole expression or some parts of the expression. An expression that does what we want might look like this:

```
"<h1>\s*([\s\S]*?)\s*</h1>"
```

Actually it doesn't quite do it but it is close. The result is a collection of matches in an object of type *MatchCollection*:

```
Item 0 .FirstIndex:89 .Length:24 .Value:"<h1>RegEx Example</h1>" .SubMatches: .Count:1 Item 0
"RegEx Example" Item 1 .FirstIndex:265 .Length:20 .Value:"<h1>Level One</h1>" .SubMatches: .Count:1
Item 0 "Level One"
```

The name of the item is in the *SubMatches* of each item but where is the text? To get that we can simply use *Mid$* together with the *FirstIndex* and *Length* properties of each match to find the start and finish of the text between the end of one *h1* and the start of the next. However, as usual there is a problem. The last match is not terminated by another *h1* element but by the end of *body* tag. So our last match will include that tag and all the stuff that can follow the body. The solution is to use another expression to get just the body first:

```
"([\s\S]*)"
```

This returns just one match with on sub-match and the sub match is everything between the body and end body tags. Now we can use our original expression on this new string and it should work.

Now that you have seen an example here is a detailed description of the expressions used and the property settings of the Regular Expression object used.

A regular expression is simply a string of characters but some characters have special meanings. In this expression:

```
"([\s\S]*)"
```

there are three principal parts:

```
""
```

```
"([\s\S]*)"
"
```

```
"
```

Each of these parts is also a regular expression. The first and last are simple strings with no meaning beyond the identity of the characters, they will match any string that includes them as a substring.

The middle expression is rather more obscure. It matches absolutely any sequence of characters and also captures what it matches. Capturing is indicated by surrounding the expression with round brackets. The text that is captured is returned as one of the *SubMatches* of a match.

"	matches just
(	begins a capture expression
[	begins a character class
\s	specifies the character class that includes all white space characters
\S	specifies the character class that includes all non-white space characters
]	ends the character class
*	means that as many instances of the preceding expression as possible are to be matched
)	ends the capture expression
"	matches

In the case studies section of this book there is a simple application that you can use to test regular expressions: Regular Expression Tester[1].

9.9 External links

- Regular Expressions in Visual Basic 6 - VB RegExp (regular-expressions.info)[2]
- VBScript's Regular Expression Support (regular-expressions.info)[3]
- .Net Regular Expression Classes (msdn.microsoft.com)[4]

1 http://en.wikibooks.org/wiki/..%2FRegular%20Expression%20Tester
2 http://www.regular-expressions.info/vb.html
3 http://www.regular-expressions.info/vbscript.html
4 http://msdn.microsoft.com/en-us/library/30wbz966%28v=vs.71%29.aspx

10 Arrays

10.1 Introduction

Arrays are extremely useful in Visual Basic, and are present in many other programming languages. Arrays are used to group similar data together, to make it easier to search and sort through this data. The best way to look at an array is to show you one. Say you want to make a phone book for your computer. Rather than make a new variable with a new name whenever you want to add a name, you can make an array. Arrays consist of the variable name and a subscript, which is put in parentheses for computer use. So, if you want to make a phone book of say, 100 people, rather than make a bunch of variables named 'person1, person2, person3....person100', you may use an array. All you have to do is dimension the array (see above). So now those 100 variables turn into person(1 to 100) in a dim statement. Now you may be asking, what do I do now? Each object stored in an array is called an element. What you have done is created an array which can hold 100 elements. If you want to print the 32nd person in your phone book, you simply would type:

```
Debug.Print Person(32)
```

10.2 Use of Arrays

Arrays have far more uses other than just making phone books. But before we get into specific uses, let's look at some of the basic techniques used when working with arrays. The first thing you may want to do is learn how to easily fill an array. Filling an array with random numbers is a popular technique for starting off programs, and testing other techniques such as sorting. To fill an array, a simple For Loop may be used.

```
Option Explicit
Dim lngIndex as Long
Dim strArray(0 to 9) as String ' Create an array with 10 values
Dim intCounter as Integer
intCounter = 1
For lngIndex = LBound(strArray) to UBound(strArray)
   intCounter = intCounter + 1
   strArray(lngIndex) = intCounter
Next lngIndex
```

The above shows an example of read-write iteration over an array. *For Each* loop can be used for read-only iteration over an array:

```
Dim MyArray(9)
For i = 0 To 9
```

```
  MyArray(i) = i
Next
For Each Item In MyArray
  Debug.Print Item
Next
```

10.3 Indices

Per default, array **indices start** at 0, unless "Option Base 1" declaration is used. Without the declaration used, an array declared as "Dim MyArray(5)" has 6 elements: 0, 1, 2, 3, 4, 5.

The **index range** of an array has to be a continuous sequence of integers, including negative numbers. Thus, the following is possible:

```
Dim MyArray1(-5 to 5)
Dim MyArray2(-10 to -5)
For i = LBound(MyArray1) to UBound(MyArray1)
   MyArray1(i) = i + 5
   Debug.Print i, MyArray1(i)
Next
For i = LBound(MyArray2) to UBound(MyArray2)
   MyArray2(i) = i + 5
   Debug.Print i, MyArray2(i)
Next
```

10.4 Size

The size of an array can be obtained using LBound and UBound as follows:

```
Dim MyArray1(-5 to 5)
Dim MyArray2(10-1)
Size1 = UBound(MyArray2) -  LBound(MyArray2) + 1
Size2 = UBound(MyArray2) + 1 'For a same array with indexing starting at zero
Debug.Print Size1, Size2
```

Keywords: length, item count, element count.

10.5 Dynamic Arrays

An array with the number of elements specified upon its declaration, as in Dim Names(0 to 9), is a static one: the number of its elements cannot be changed in runtime. By contrast, an array declared without the number of elements, as in Dim Names(), is a *dynamic array*, and its number of elements can be changed using ReDim. To preserve the element content of the array when using ReDim, Preserve keyword has to be used after ReDim.

Say you have a phone book program running and it has an array of your friends' names, like this:

```
Option Explicit
Dim StrNames(0 to 2) As String
```

```
StrNames(0) = "Alec"
StrNames(1) = "Jack"
StrNames(2) = "Pie"
```

But say you want to add *more* friends to your phone book, when a button is clicked, for example. What do you do? You can **define a dynamic array** instead:

```
Dim Names() As String
```

This array initially has no elements. You can add more elements, by using the ReDim keyword:

```
ReDim Names(0 to 1) As String
```

This would create two new elements. You can assign names to these elements in the same way as before:

```
Names(0) = "Alec"
Names(1) = "Jack"
```

If you want to add more entries then use, for example:

```
ReDim Names(0 to 2) As String
```

But you'll end up losing all the old records! If you want to keep them, you must also use the Preserve keyword:

```
ReDim Preserve Names(0 To 3) As String
Names(1) = "Eugene H / Orage" 'Whoa this person's name is too long to remember
```

To iterate AKA **loop over** the elements of a **dynamic array**, you have to use LBound and UBound functions or For Each loop, as described at #Use of Arrays[1].

To **add an element** to an initialized dynamic array, you can proceed as follows, using UBound function:

```
ReDim Preserve Names(0 To UBound(Names) + 1) As String
Names(UBound(Names)) = "my new friend"
```

However, UBound does not work with an un-initialized dynamic array, so the following would cause a run-time error:

```
Dim MyArray() As String
Debug.Print UBound(MyArray)
```

One way to avoid this run-time error is to keep a flag:

```
Dim ArrayIsEmpty As Boolean
Dim MyArray() As String

ArrayIsEmpty = True

If Not ArrayIsEmpty Then
    ReDim Preserve MyArray(0 To UBound(MyArray) + 1) As String
Else
```

1 Chapter 10.2 on page 59

```
    ReDim Preserve Names(0 To 0) As String
    ArrayIsEmpty = False
End If
Names(UBound(Names)) = "my new friend"
```

Another way to avoid the run-time error is to catch the error created by the use of "UBound" on an uninitialized array:

```
Dim MyArray() As String

NumberOfElements = 0
On Error Resume Next
NumberOfElements = UBound(MyArray) + 1
On Error GoTo 0
ReDim Preserve MyArray(0 To NumberOfElements) As String
MyArray(UBound(MyArray)) = "my new friend"
```

Finally, you can use the following *hack* to check if a dynamic array has been initialized:

```
Dim MyArray() As String

If (Not MyArray) = -1 Then
  NumberOfElements = 0 'The array is uninitialised
Else
  NumberOfElements = UBound(MyArray) + 1
End If
ReDim Preserve MyArray(0 To NumberOfElements) As String
MyArray(UBound(MyArray)) = "my new friend"
```

10.6 Variant Arrays

Variant arrays are dynamic arrays declared using the Variant type, and initialized using "= Array()". Their advantage is that, after they are initialized using "= Array()", LBound and UBound functions work with them even when they have no elements, as follows:

```
Dim VariantArray As Variant

VariantArray = Array()

Debug.Print LBound(VariantArray) 'Prints 0
Debug.Print UBound(VariantArray) 'Prints -1
```

As a consequence, adding an element is straightforward, with no need to check for UBound failure:

```
ReDim Preserve VariantArray(0 To UBound(VariantArray) + 1) As Variant
VariantArray(UBound(VariantArray)) = "Jennifer"
```

Variant arrays can be initialized to a set of values:

```
Dim VariantArray As Variant
VariantArray = Array(1, 2.3, "Hey")
```

For very large arrays, the overhead of using variant arrays, which have the element type of Variant rather than String or other narrower type, may well be restrictive, in which case

conventional dynamic arrays should be used. What seems also much faster than variant arrays for adding items one at a time are ../Collections/[2].

10.7 Multi-Dimensional Arrays

Arrays can be defined to have any number of dimensions (or indices), by listing the sizes of each dimension:

```
Dim FunkyArray(2,3,1,4) As String
```

An element can be referenced like this:

```
FunkyArray(1,2,0,3) = 24
```

Dynamic arrays can also be re-dimensioned to have any number of dimensions:

```
Dim VeryFunkyArray() as String

ReDim VeryFunkyArray(2,2,2) As String
```

The LBound and UBound functions can be used to find the bounds of a particular dimension:

```
Debug.Print UBound(FunkyArray, 4)
```

Would give 4.

Using UBound without the second parameter gives the first dimension

```
Debug.Print UBound(FunkyArray)
' Displays 2
Debug.Print UBound(VeryFunkyArray)
' Displays 2
```

10.8 Erasing Arrays

Any type of array can be re-set to empty by using:

```
Erase SomeArray
```

10.9 Mixing Arrays

The real power of arrays comes when defining arrays of arrays. What does this mean? Declare an array:

```
Dim VariantArray() As Variant

VariantArray = Array()
```

2 Chapter 10.11 on page 65

```
ReDim VariantArray(1) As Variant

VariantArray(0) = Array(1,2,3)
VariantArray(1) = Array(4,5,6)
```

What do we have here? Essentially two arrays inside another. They can be referenced like this:

```
Debug.Print VariantArray(0)(2)
```

Would show 3.

You can nest arrays like this to any depth and in any order and they can be of any size. A note of caution must be taken when using the ReDim statement, however. You cannot specifically re-dimension a particular dimension of an array; instead you need to temporarily copy to a variant.

```
Dim vtemp As Variant

vtemp = VariantArray(0)

ReDim vtemp(1+UBound(vtemp)) As Variant
vtemp(UBound(vtemp)) = 7

VariantArray(0) = vtemp
```

10.10 Use of Matrices

See also the section #Multi-Dimensional Arrays[3]*.*

Matrices are not as commonly used as arrays, but are an important element of programming. Rather than just one dimension, a matrix may have 2 or more. So, to make a matrix, the Dim statement would be:

```
Dim Matrix(0 To 9, 0 To 9) as Integer
```

This will create a matrix that is 10 by 10 and comprised of integers. In reality, a matrix is much like an array of arrays. The first thing you'll want to do is know how to create a loop for reading and filling a matrix. But before that, even, you should know how matrices are structured. An example matrix would be:

```
    1 2 3 4 5 6 7 8 9

1   1 1 1 1 1 1 1 1 1
2   1 1 1 1 1 1 1 1 1
3   1 1 1 1 1 1 1 1 1
4   1 1 1 1 1 1 1 1 1
```

This is a 4 by 9 matrix. When talking about matrices, rows are always stated before columns. You will also notice column numbers travel from left to right, while row numbers

3 Chapter 10.7 on page 63

travel from top to bottom. This is very important. And of course, matrices need not consist of merely 1s. So, now you want to fill a matrix? Its very similar to an array.

```
For Row = 0 To 3
    For Column = 0 To 8
        Matrix(Row, Column)=1
    Next
Next
```

This will of course fill the matrix with nothing but ones, but random numbers may be used (see references). As you may notice, this involves nested loops. The loop fills left to right, top to bottom (the same directions we use when reading).

10.11 Sorting

For sorting variant arrays, see `http://stackoverflow.com/questions/152319/vba-array-sort-function`.

11 Collections

Collection class provides an array-like container more flexible than an array in some ways and less flexible in other ways. More flexible in some ways than Collection is Dictionary class.

Creating a new collection:

```
Dim Cats As Collection
Set Cats = New Collection
```

Adding an item:

```
Cats.Add "Item"
Cats.Add "Item", "Key"
...
```

Accessing an item at an index, in a read-only way:

```
Cats (3) 'The third member of the collection
Cats.Item(3) 'An alternative
Cats.Item("Key 3") 'Works if an item has a key associated
```

Overwriting a item at an index:

```
NewValue = MyCollection(i) + 1
MyCollection.Add NewValue, Before:=i
MyCollection.Remove Index:=i + 1
```

Removing an item:

```
Collection.Remove Index
Collection.Remove HashKey
```

The size of a collection:

```
Cats.Count
```

Iterating over a collection, read-only:

```
For Each Cat In Cats
   Rem Do something on Cat
Next
```

Iterating over a collection, read-write:

```
'Fill the collection
Set MyCollection = New Collection
For i = 1 To 10
  MyCollection.Add i
Next
```

```
'Increment each item by 1
For i = 1 To MyCollection.Count
  NewValue = MyCollection(i) + 1
  MyCollection.Add NewValue, Before:=i
  MyCollection.Remove Index:=i + 1
Next
```

Testing the emptiness of a collection:

```
If Cats.Count=0 Then
  '...
End If
```

Testing the presence of an element in a collection:

```
MatchFound = False
For Each Member In MyCollection
  If Member = "SoughtValue" Then
    MatchFound = True
    Exit For
  End If
Next
```

Appending one collection to another:

```
For Each Member In AppendedCollection
  ExtendedCollection.Add Member
Next
```

Converting a collection to an array:

```
Dim MyArray
ReDim MyArray(MyCollection.Count - 1)
Index = 0
For Each Member In MyCollection
  MyArray(Index) = Member
  Index = Index + 1
Next
```

Using a collection as a queue:

```
Set Queue = New Collection
For i = 1 To 10
  Queue.Add i
Next
For i = 1 To 5
  Queue.Remove 1 'Remove from the queue
Next
```

Using a collection as a stack:

```
Set Stack = New Collection
For i = 1 To 10
  Stack.Add i
Next
For i = 1 To 5
  Stack.Remove Stack.Count 'Remove from the stack
Next
```

Using a collection as a set:

```
' Using the key of a collection item will do the trick.
```

```
' The key has to be a string, hence "Str(i)" below.
Set NumberSet = New Collection
For i = 1 To 10
  NumberSet.Add i, Str(i) 'Value, Key
Next
For i = 5 To 15
  'Add while catching errors resulting from existence of the key
  On Error Resume Next
  NumberSet.Add i, Str(i) 'Value, Key
  On Error GoTo 0
  'End If
Next
For i = 14 To 16
  'Remove
  On Error Resume Next 'Catch error if element not present
  NumberSet.Remove Str(i)
  On Error GoTo 0
Next
Set NumberSet2 = New Collection
For i = 10 To 25
  NumberSet2.Add i, Str(i)
Next
'Union
Set SetUnion = New Collection
For Each Item In NumberSet
  SetUnion.Add Item, Str(Item) 'Value, Key
Next
For Each Item In NumberSet2
  On Error Resume Next
  SetUnion.Add Item, Str(Item) 'Value, Key
  On Error GoTo 0
Next
'Intersection
Set SetIntersection = New Collection
For Each Item In NumberSet
  On Error Resume Next
  Dummy = NumberSet2(Str(Item))
  If Err.Number = 0 Then
    SetIntersection.Add Item, Str(Item) 'Value, Key
  End If
  On Error GoTo 0
Next
'Set difference
Set SetDifference = New Collection
For Each Item In NumberSet
  On Error Resume Next
  Dummy = NumberSet2(Str(Item))
  If Err.Number <> 0 Then
    SetDifference.Add Item, Str(Item) 'Value, Key
  End If
  On Error GoTo 0
Next
```

12 Dictionaries

Dictionary class provides some functions beyond those provided by Collection class. A dictionary is a list of key, value pairs. Unlike Collection, Dictionary class is only available via Microsoft Scripting Runtime, a reference to which needs to be added to your project in order to use the class.

Creating a dictionary:

```
Set Dict = New Dictionary
Set Dict = CreateObject("Scripting.Dictionary") 'An alternative
```

Adding a key and an item to a dictionary:

```
Dict.Add "Key1", "Value1"
Dict.Add "Key2", "Value2"
Dict.Add Key:="Key3", Item:="Value3"
```

Accessing an item by key:

```
Value3 = MyDict.Item("Key3")
```

Accessing an item by index:

```
Index = 2
Counter = 1
For Each Key In Dict.Keys
  If Counter = Index Then
    FoundKey = Key
    FoundValue = Dict.Item(Key)
    Exit For
  End If
  Counter = Counter + 1
Next
```

Iterating through the keys:

```
For Each Key In Dict.Keys
  Value = Dict.Item(Key)
Next
```

Iterating through the values:

```
For Each Value In Dict.Items
  '...
Next
```

Removing an item:

```
Dict.Remove "Key3"
```

Removing all items or emptying:

```
Dict.RemoveAll
```

Size or number of elements:

```
Dict.Count
```

Testing for emptiness:

```
If Dict.Count = 0 Then
  '...
End If
```

Changing the key of an item:

```
Dict.Key("Key1") = "Key1a"
```

Changing the value of an item:

```
Dict.Item("Key2") = "Value2a"
```

Testing for presence of a key:

```
If Dict.Exists("Key2") Then
  '..
End If
```

Using dictionary as a set:

Associate the dummy value 1 with each element of the set.

When adding an item, test for existence.

```
Set DictSet = New Dictionary
DictSet.Add "Item1", 1
'DictSet.Add "Item1", 1 -- error: the item is already there
If Not DictSet.Exists("Item1") Then
  DictSet.Add "Item1", 1
End If
DictSet.Remove "Item1"
```

13 Data Types

Data types in Visual Basic can be divided into three groups:

- **Native**: Types that are understood directly by the Visual Basic compiler without assistance from the programmer
- **User-defined**: commonly referred to by the initials UDT, meaning User defined Type, these correspond to Pascal records or C structs
- **Classes**: the basis for object oriented programming in Visual Basic. Classes include forms, add-ins, and database designers.

13.1 Built in Types

The built in types are:

Byte

8 bit, unsigned

Integer

16 bit, signed

Long

32 bit signed

Single

32 bit floating point, range about $\pm 10^{38}$

Double

64 bit IEEE floating point, range about $\pm 10^{308}$

Currency

exact representation of decimal numbers of up to four decimal places

String

dynamically allocated UniCode strings, theorectical capacity about 2^9 characters.

Collection

an associative array of Variants.

Date

8 byte date/time value range January 1, 100 to December 31, 9999

Object

a holder for any type of Object.

Variant

a holder for any type of value or object.

13.2 Byte, Integer & Long

Example:

```
Dim a as Byte
Dim i as Integer
Dim x,y as Long 'Define two variables. Note that only the last variable will
be a long integer.
```

Now those variables will only be capable of storing integer values (without decimal). Long integers can store a number with a bigger range of value than integers but they occupy a bigger space of RAM.

Type	Storage	Range of Values
Byte	1 byte	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long	4 bytes	-2,147,483,648 to 2,147,483,647

Some functions you need to know: Int()

Int() converts a decimal value into an integer value:

```
Dim i as Integer
i=Int(3.9)
Print i 'Prints 3
```

13.3 Single & Double

These data types can store decimal values. "Double" compared to "Single" is similar to the "Long" compared to "Integer":

Type	Storage	Range of Values
Single	4 bytes	-3.402823E+38 to -1.401298E-45 for negative values1.401298E-45 to 3.402823E+38 for positive values.
Double	8 bytes	-1.79769313486232e+308 to -4.94065645841247E-324 for negative values4.94065645841247E-324 to 1.79769313486232e+308 for positive values.

Some useful functions: Round()

Round() rounds off a decimal to a certain number of decimal digits that the programmer wants. The first argument should be a decimal value which you want to round off. The second argument specifies the number of decimal digits you want, for example:

```
Dim pi as Double
pi=3.141592653589
pi=Round(pi,2) 'Rounds off 3.141592653589 to only two decimal digits
Print pi 'Prints 3.14
```

13.4 String

A string is an array of characters. As an example:

```
Dim a As String
a = "This is a string"
```

Strings can be concatenated (connected together to form a new string) using the "&" operator. For example,

```
dim b as String
b = "Wiki" & "book" & "s"
Print b 'Prints "Wikibooks"
```

A normal string variable occupies 10 bytes of RAM, plus the string's size, and can hold up to 2 billion characters!

Some frequently used built-in string constants: vbTab, vbCrLf

vbTab contains a string that does the same thing as the Tab key on your keyboard, while vbCrLf creates a character return and a line feed(similar to the Enter key):

```
Print "Jack:" & vbTab & "1 pie" & vbCrLf & "me:" & vbTab & "10 pies"
```

Will print:

```
Jack:    1 pie
me:      10 pies
```

To include special characters and quotation marks in the strings, the Chr() function may be used:

```
Dim a As String
Print "A quotation mark: [" & Chr(34) & "]"
a = "Replace 'apostrophes' for quotation marks"
Replace( a, "'", Chr(34) )
Print a
```

Some string functions: Str(),Val(),inStr(),Mid(),Replace(),Trim()

In fact there are tons of built-in string manipulation functions available. But right now, I'll just introduce two: Str() and Val().

Str() converts any numerical value into a string value while Val() converts a string value into a numerical value(only when it's convertible).

```
Dim MyString As String
Dim MyNumber As Single
MyString=Str(300) 'converts a number into a string
MyNumber=Val("300") 'converts a string into a number
```

Even if you don't do the conversion, you will not end up getting Type Mismatch Errors. However, it is considered better to use explicit type conversions, because it is easier to debug.

Even if you do be prepared for next to impossible to debug problems caused by VB refusing to convert something and refusing to tell you what the heck it is. VB is extremely touchy and raises an exception at the least expected times.

13.5 Structure

An example definition of a structured type:

```
Type E2Point
  x As Double
  y As Double
End Type
Sub Test()
  Dim MyPoint As E2Point
  Dim MyPoint2 As E2Point
  MyPoint.x = 4
  MyPoint.y = -5
  MyPoint2 = MyPoint 'Make a copy
  MyPoint2.x = 3
  MyPoint2.y = -6
  Debug.Print MyPoint.x, MyPoint.y '4, -5: not overriden with 3 and -6
  Debug.Print TypeOf MyPoint Is E2Point 'True
  Debug.Print TypeOf MyPoint Is Object 'False
End Sub
```

The type has to be defined outside of a procedure.

A variable of a structure type is not an object.

Links:

- Type Statement[1], Office 2013, msdn.microsoft.com
- Type Statement[2] at Visual Basic for Applications Reference, msdn.microsoft.com

13.6 Enumeratation

An example definition of an enumerated type:

1 http://msdn.microsoft.com/en-us/library/office/gg278730.aspx
2 http://msdn.microsoft.com/en-us/library/aa266315.aspx

```
Enum Colors
  Red '=0
  Green '=1
  Blue '=2
End Enum
Enum Colors2
  Red2 = 1
  Green2 '=2
  Blue2 '=3
End Enum
Sub Test()
  Debug.Print Red, Green, Blue
  Debug.Print Red2, Green2, Blue2
  Dim MyVar As Colors 'Ends up being typed as Long
  MyVar = 8 'Does not lead to an error: no restriction on values
End Sub
```

Links:

- Enum Statement[3], Office 2013, msdn.microsoft.com
- Enum Statement[4] at Visual Basic for Applications Reference, msdn.microsoft.com

13.7 Type Test

To find out about the type of a variable, you can use "**TypeOf** ... **Is** ..." test. The tested types can only be object types and structure types, but the test can be applied to any variable, whether typed as an integer, a string or an object.

An example of TypeOf in Excel:

```
Set MyVar = Selection
Debug.Print "Selection is an object: " & TypeOf MyVar Is Object
Debug.Print "Selection is a range: " & TypeOf MyVar Is Range
Debug.Print "Sheets is of type Sheets: " & TypeOf Sheets Is Sheets
MyStr = "Hello"
Debug.Print "Text is an object: " & TypeOf MyStr Is Object
If TypeOf MyVar Is Range Then
  Set MyCells = MyVar.Cells
End If
```

An example test with a "Select Case True":

```
Set MyVar = new Collection
Select Case True
Case TypeOf MyVar is Range
  Debug.Print "Range"
Case TypeOf MyVar is Collection
  Debug.Print "Collection"
Case Else
  Debug.Print "Other cases"
End Select
```

You can further find out about a type using **IsObject** and **TypeName** functions:

```
Debug.Print IsObject(Selection) 'True
```

3 http://msdn.microsoft.com/en-us/library/office/gg251516.aspx
4 http://msdn.microsoft.com/en-us/library/aa243358.aspx

```
Debug.Print IsObject("Hello") 'False
Debug.Print TypeName("Hello") 'String
Debug.Print TypeName(4) 'Integer
Debug.Print TypeName(3.5) 'Double
Debug.Print TypeName(Selection) 'Range
```

Links:

- If...Then...Else Statement[5] at Visual Basic for Applications Reference, msdn.microsoft.com
- TypeName Function[6] at Visual Basic for Applications Reference, msdn.microsoft.com
- IsObject Function[7] at Visual Basic for Applications Reference, msdn.microsoft.com
- IsArray Function[8] at Visual Basic for Applications Reference, msdn.microsoft.com
- IsDate Function[9] at Visual Basic for Applications Reference, msdn.microsoft.com
- IsNumeric Function[10] at Visual Basic for Applications Reference, msdn.microsoft.com
- IsNull Function[11] at Visual Basic for Applications Reference, msdn.microsoft.com

5 http://msdn.microsoft.com/en-us/library/aa243382.aspx
6 http://msdn.microsoft.com/en-us/library/aa263394.aspx
7 http://msdn.microsoft.com/en-us/library/aa445054.aspx
8 http://msdn.microsoft.com/en-us/library/aa445040.aspx
9 http://msdn.microsoft.com/en-us/library/aa445042.aspx
10 http://msdn.microsoft.com/en-us/library/aa445052.aspx
11 http://msdn.microsoft.com/en-us/library/aa445050.aspx

14 Procedures and Functions

Functions are named blocks program code that perform a specific task and return a result. The task can be as simple as adding two numbers or as complex as launching a spacecraft. A subroutine is like a function, just that it does not return a result.

14.1 Defining procedures

An example function definition:

```
 Public Function Sum(ByRef Number1 As Double, ByRef Number2 As Double) As
Double
    'Return the result by writing it to a variable having the same name as the
function
   Sum = Number1 + Number2
 End Function
```

An example subroutine definition:

```
 Public Sub Tell(ByVal MyString1 as String, ByVal MyString2 as String)
   MsgBox MyString1 & MyString2
 End Sub
```

Note the following:

- The arguments to the function are declared as ByRef[1] which requires the compiler to make sure that only arguments of the specified type are used, in this case Double.
- The function returns a value by assigning it to the function name as though the function were a variable. This contrasts with the use of the keyword *return* in many other languages.

14.2 Calling procedures

You can use or call the two procedures defined in the previous sections as follows:

```
 'On the next line, argument order matters
 Tell "Hello there.", ""
 'On the next line, names of arguments are used and argument order does not
matter.
 Tell MyString1:="Hello there,", MyString2:=" again."
 'Tell ("Hello there.","") -- syntax error

 MySum = Sum(123.456, 234)
```

1 Chapter 33.9 on page 237

```
MySum2 = Sum(Number2:=8, Number1:=6)
'MySum3 = Sum Number2:=8, Number1:=6 -- syntax error
```

Note the following:

- The arguments (*argument list*) passed to a *function* must be enclosed in round brackets, whereas those supplied to a subroutine need not.

14.3 Procedure parts

Each function and subroutine has the following parts, some of the optional:

Visibility

Public, *Friend* or *Private*

Procedure Type

Sub, *Function*, *Property Let*, *Property Get*, *Property Set*

Name

A freely chosen name that starts with a letter and contains only letters, numbers and underscores.

Argument List

A list of the items of data that the procedure reads or writes into.

Return Type

For a *Function* or *Property Get*, the data type returned, such as *Double* or *String*.

Body

All the statements that do the work.

Only the *Name* and the *Procedure Type* are mandatory. Of course, a procedure without a body doesn't do anything.

14.4 Visibility

This seems like a very unimportant part of a procedure declaration to most people but in fact it is a very helpful feature. With it you can show that some procedures are just for use inside a module (*Private*), some only for use in this component (*Friend*) or available for the whole world (*Public*). You should mark procedures *Private* unless they will be called from outside the *module*. this will encourage you, and anyone who edits your program, to place related procedures in the same module which obviously makes maintenance easier.

Marking a procedure *Private* also means that you can have another procedure with exactly the same name in another module.

14.5 Early Termination

Use **Exit Function** or **Exit Sub** to terminate a procedure in the middle, like this:

```
Sub LengthyComputation(fat, n)
  If n = 0 Or n = 1 Then
    fat = 1
    ' Now terminate
    Exit Sub
  End If
  ' Now compute fat ...
End Sub
```

14.6 Procedure type

All procedures are either functions that return a result as the value of the function, or subroutines that are called for their side effects. To return a value, you can use both, but with subroutine, you need to do it via an argument:

```
Private Function FunctionHalf(ByRef y as Double) as Double
  FunctionHalf = y / 2
End Function

Private Sub SubroutineHalf(ByRef y As Double, ByRef Result As Double)
  Result = y / 2
End Sub
```

The two procedures do essentially the same thing, that is, divide a number by two. The *Function* version does it by *assigning* the new value to the name of the function while the *Sub* version assigns it to the name of one of the *arguments*. This affects how you use them.

The function version can be used in an expression as follows:

```
Debug.Print FunctionHalf(10) 'Prints 5
```

To use the subroutine to return value, you need to store the value in a variable, like this:

```
Dim nHalf as Double
SubroutineHalf 10, nHalf
Debug.Print nHalf
```

Generally, you use a *Function* when the result is a single thing (number, string, object) and a *Sub* when you either want to return several distinct things or nothing at all.

Properties are also a form of procedure. *Property Get* is a function; *Property Let* and *Property Set* are subroutines. For more discussion of *Properties*, see the ../Object Oriented Programming/[2] chapter.

2 Chapter 23.2 on page 114

14.7 Optional arguments

You can specify optional arguments and default values:

```
Function MySum(i1, i2, Optional i3)
  If IsMissing(i3) Then
    MySum = i1 + i2
  Else
    MySum = i1 + i2 + i3
  End If
End Function

Function MyConcatenate(String1, String2, Optional Separator = " ")
  'Default value for Separator specified
  MyConcatenate = String1 & Separator & String2
End Function

Sub Test()
  Debug.Print MySum(1, 2) * MySum(1, 2, 3)
  Debug.Print MyConcatenate("Hello", "World")
  Debug.Print MyConcatenate("Hello", "World", ", ")
End Sub
```

Once an argument is declared optional, all arguments at the right of it have to be declared optional as well.

Links:

- Using Optional Arguments[3], Office 2000, msdn.microsoft.com

3 http://msdn.microsoft.com/en-us/library/office/aa164532.aspx

15 Error Handling

Error handling in Visual Basic, an outline:

- On Error Goto <Label>
- On Error Goto 0
- On Error Resume Next
- If Err.Number = 0
- If Err.Number <> 0
- Resume
- Resume <Label>
- Resume Next
- Err.Description
- Err.Raise <Number>

Suppressing the error and detecting it using the non-zero error number:

```
Set MyCollection = New Collection
MyCollection.Add "Item", "Item"
On Error Resume Next
MyCollection.Add "Item", "Item" 'This result in an error
MyErrNumber = Err.Number
On Error Goto 0 'Restore the absence of error handling
If MyErrNumber <> 0 Then
  'Error occurred
  MsgBox "Item already present in the collection."
End If
```

Creating an error handler:

```
Sub Test()
  On Error Goto ErrorHandler
  ...
  Exit Sub

ErrorHandler:
  ...
End Sub
```

Creating an error handler that differentiates per error number:

```
Sub Test()
  On Error Goto ErrorHandler
  ...
  Exit Sub

ErrorHandler:
  Select Case Err.Number
  Case 0
    'No error
  Case 5
    '...
```

```
  Case Else
    '...
  End Select
End Sub
```

See also ../Effective Programming#Errors_and_Exceptions[1] and ../Coding_Standards#Error_Handling[2].

1 Chapter 25.10 on page 134
2 Chapter 31.37 on page 213

16 Files

Another essential part of Visual Basic is file Input/Output, that is, working with files. While programming, you may want at some point to save data so they may be accessible for further use. This is where file I/O comes in. VB allows us to perform most operations available in Windows Explorer and DOS command line. Before you can do this, you need to understand how file I/O works.

16.1 Layout of Data

VB generally arranges data into records and fields for the programmer's convenience. The best way to explain this is by illustrating it:

```
Bob,Larry,George
Kevin,Ken,Alec
```

This is obviously data containing names. Each line of data is called a record. So the first record contains Bob, Larry, and George. A field is each item in a record. So the first field in the first record would be Bob, the second field would be Larry, and so on. There may be as many fields and records as you see fit.

Records and fields are created really just for the programmer's convenience. For greater flexibility, you can also view the whole thing as a string. And you can split the string into records and fields yourself with the built-in Split() function.

16.2 Input

When a program receives data from an outside source, it is considered input. In Visual Basic, this source is usually a standard text file, with a .txt file extension(as which could be read with Notepad). First, you need to open the file from which you want to read the data. This is done using the following code:

```
Open <filename> For <mode> As <channelnumber>
```

For example:

```
Open "c:\filename.txt" For Input As #1
```

The file path can be anything you want, if it doesn't exist, a new file (and directory(s)) will be created. The extension of the file doesn't matter much. It will not affect the content of the file nor the process of writing/reading the file. So you can also do this:

```
Open "c:\filename.myfile" For Input As #1
```

Or this:

```
Open "c:\filename" For Input As #1
```

The open and file path are self explanatory. However, "for input" states it is being used to receive data from an outside file, VB is "inputting" data from a file into variables . When outputting(writing to a file), this is changed to "for output". When we want to open a file in binary, it's "for binary". "As #1" is which channel is being used. When you are seeking specific information from the file you would use "for random". A channel could be seen as Visual Basic setting up a pathway to the file. Only one file can be used on a channel at a time. So if you want to open multiple files without closing any, you can use different channels. To input a line of data, it is fairly simply:

```
Input <channel>, <variables>
```

For example:

```
Input #1, x, y, z
```

This will read data from whatever channel 1 is connected to. It will store the first field of the current record into X, and the second field into Y, and the third into Z.

Finally, we should ALWAYS close the file after the work is done. The syntax is simple:

```
Close <channel>
```

As an example:

```
Close #1
```

More on using input will be covered in the "Use of Data" section.

Note: If the code will be part of a larger program, the channel number should be obtained in a dynamic way instead of a hard-coded number. This is done using the function **FreeFile()**

```
  Dim nLogFile As Long
nLogFile = FreeFile()
Open "c:\filename" For Input As #nLogFile
...
Close #nLogFile
```

16.3 Output

Output is very similar to input. The only difference is, well, it sends information OUT of the program and into a file. Once again, we shall only use text files (.txt). First you must open the file:

```
Open "C:\filepath.txt" For Output As #1
```

You may have noticed, the only thing that differs from opening a file for input, is the "For Output". Once this is done, everything else is simple.

```
Write #1, x, 5
```

Assuming X = 3, if you open the text document it will look like:

```
3, 5
```

You can write any value or variable to a file, however it is important to note that, when writing a string, it should be placed in quotes. If it is a variable containing a string, this is done automatically. Another important note is that two consecutive write statements will write to separate lines. For example:

```
Write #1, 5, 4
Write #1, 7, 4
Write #1, "this is a string"
```

Will appear in the file as

```
5, 4
7, 4
"this is a string"
```

Keep in mind, when outputting to a file, it will ERASE all prior information in the text. To add on to existed information, the "Append" command must be used.

If you are writing data to a file for use by some other program, you may find that you don't want the quotes that VB puts around your strings. You can avoid writing the quotes by using the Print command instead of the Write command. For example:

```
Print #1, "this is a string"
```

Will appear in the file as

```
this is a string
```

If, during runtime, the file at hand was modified, and a new line was added which contained necessary data, and the open statement ended as "#1", whatever was in line 1 of the file would be written over(only if the file was opened as Output). To solve this, a variable is dimmed as FreeFile (Dim FF as FreeFile). This will open the first free slot or line in the file at hand. The new Open statement would become:

```
Open "C:\filepath.txt" For Output As FF
```

(will erase file)

```
Open "C:\filepath.txt" For Append As FF
```

(will **not** erase file)

Using the second statement, the first free line in the file will be written to.

16.4 How to specify the path to a file

In practice it is rare for open statements to look like this:

```
Open "C:\filepath.txt" For Append As FF
```

Usually the path to the file will be held in a *string* variable:

```
Dim s As String
s = "C:\filepath.txt"
Open s For Append As FF
```

The variable, *s*, simply needs to hold a path to the file. The path can be an absolute path such as:

```
s = "d:\users\me\my file.txt"
```

or a file in the current directory (use *CurDir* to find out what that is):

```
s = "my file.txt"
```

or a path relative to the current directory like this:

```
s = "..\you\your file.txt"
```

or like this:

```
s = "mysub\myother file.txt"
```

For the more adventurous it can also include a reference to a shared file on a server that doesn't use a drive letter:

```
s = "\\someserver\someshare\somefile.txt"
```

16.5 Examples

An example form using file reading can be found here.[1] Right click 'here' in Internet Explorer and then 'Save Target As...' to save the form file.

Please upload more files with examples. It will be really helpful for the developer.

1 http://www.freewebs.com/3zzzpmk/TextView.frm

17 User Interfaces

A user interface is the part of your program that is visible to a human user. It can be as simple as a command line[1] or as sophisticated as an immersive virtual reality simulator (see Death Dream by Ben Bova[2]). However, in the context of Visual Basic it usually means what is commonly referred to as a Graphical User Interface or GUI which generally consists of one or more Forms that contain text boxes, labels, buttons, picture boxes, etc.

A more common example of a GUI would be a standard EXE(Windows Forms[3]).

1 http://en.wikibooks.org/wiki/Guide%20to%20Windows%20commands
2 http://www.ibdof.com/IBDOF-book-detailedview.php?book_id=5023
3 http://en.wikipedia.org/wiki/Windows_Forms

18 Simple Graphics

This chapter is a simple introduction to drawing simple graphics in Visual Basic Classic. It is possible to do very sophisticated graphics programming in VB but here we will concentrate of the basics.

There are two ways of creating simple two dimensional graphics in VB:

- draw lines, boxes, circles, and ellipses at run time by executing statements or
- add shape objects to the form at design time.

If you know in advance exactly what to draw, then it might be simpler to use the second method, as you can then see exactly what you will get as you drag and drop shapes to the form. However if the picture is to be generated based on, say, some statistics, you might be better off using the drawing methods of a *Form* or *PictureBox*.

18.1 Shape and Line controls

There are two built-in controls in Visual Basic that can be used to create simple two-dimensional drawings:

- Shape
- Line

Here is a simple example showing some of the shapes that can be created simply by dropping Shape controls on a Form:

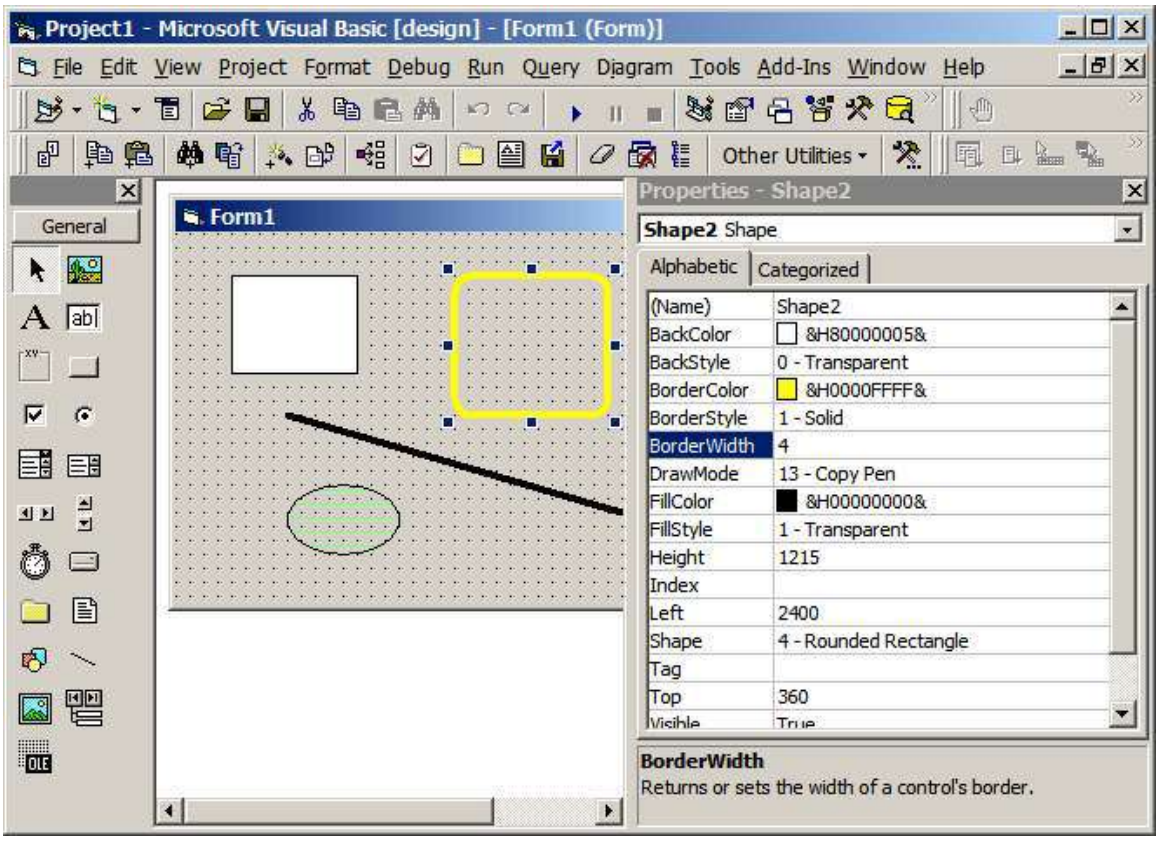


Figure 3

Note that you can modify the properties of Shape and Line controls at run time just as you can any other control. You can also create them at run time, either by using *Control Arrays* or by using the *CreateObject* function.

19 Windows Dialogs

Windows dialogs are useful when you would like to retrieve information from the user, as in asking where to save a file, letting the user to choose a color or font and what settings to use when printing (copies, what printer to use, ect.). This saves you from a lot of unnecessary work reinventing the wheel - as redesigning and making dialogs can be quite time-consuming - but is also essential in establishing a standard user interface across applications. Standardization of different aspects of the user interface, both in terms of normal widgets[1] as well as common ways of combining them, ensures that applications are intuitive - i.e. that once someone has learned to use an application, this knowledge can be employed to other applications as well.

The easiest way to access these controls is to use a component called *Microsoft Common Dialog Control 6.0*. To add this component to your project, choose *Project - Components*, and mark it in the controls list. Subsequently, a new control should appear in your control toolbox. Choose it, and place the control onto your form (note it's only visible in design time).

The control allows the use of the following dialogs. The flags mentioned is different values that can be set to the Flags property, changing the behaviour or design of the different dialogs.

19.1 Open and Save dialogs

ShowOpen shows a dialog that lets the user choose a drive, directory, file name and file extension to (presumably) open a file.

The save dialog is identical in apperance and function, with the exception of the caption. At run time, when the user chooses a file and closes the dialog box, the *FileName* property is used to get the selected file name.

1 `http://en.wikipedia.org/wiki/widget_%28computing%29`

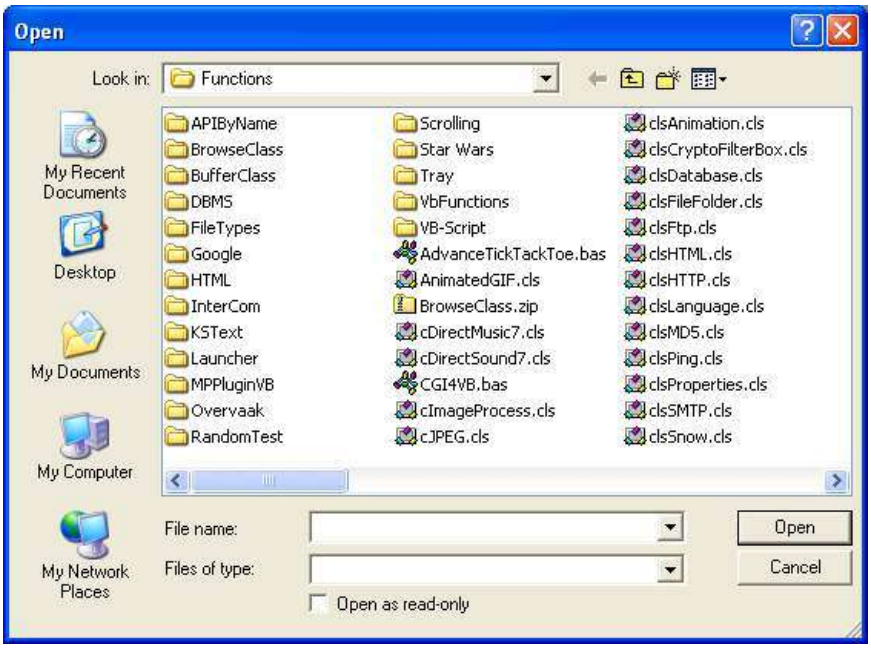


Figure 4

19.1.1 Before displaying the dialog

Before you can use these dialogs, you must first set the Filter property to allow the user to select a file extension (set default extension using *DefaultExt*). The filter property is a string with the following structure:

```
description1|filter1|description2|filter2|
```

Here, the description is the string that shows up in the list box of all the available filters to choose from. The filter is a file glob expression (otherwise known as a wild card expression) that does the actually filtering, for example, *"*.txt"*, which disallows any file extensions but TXT. It is important to recognize the difference between expressions of this kind and regular expressions which are much more powerful but also more complicated, see ../Regular Expressions/[2].

19.1.2 Code example

```
On Error Resume Next

' Clear errors
Err.Clear

' Use this common dialog control throughout the procedure
```

2 Chapter 8.10 on page 51

```
    With CommonDialog1

        ' Raises an error when the user press cancel
        .CancelError = True

        ' Set the file extensions to allow
        CommonDialog1.Filter = "All Files (*.*)|*.*|TextFiles
(*.txt)|*.txt|Batch Files (*.bat)|*.bat"

        ' Display the open dialog box.
        .ShowOpen

        ' Ignore this if the user has canceled the dialog
        If Err <> cdlCancel Then

            ' Open the file here using FileName
            MsgBox "You've opened '" & .FileName & "'"

        End If

    End With
```

19.2 Color dialog

The color dialog allows the user to select a color from both a palette as well as through manual choosing using RGB or HSL values. You retrieve the selected color using the Color property.

Figure 5

19.2.1 Code example

```
' Don't raise errors normally
On Error Resume Next

' Clear errors
Err.Clear

' Use this common dialog control throughout the procedure
With CommonDialog1
```

```
        ' Raises an error when the user press cancel
        .CancelError = True

        ' Set the Flags property.
        .Flags = cdlCCRGBInit

        ' Display the Color dialog box.
        .ShowColor

        ' Ignore this if the user has canceled the dialog
        If Err <> cdlCancel Then

            ' Set the form's background color to the selected color.
            Me.BackColor = .Color

        End If

    End With
```

19.3 Font dialog

The Font dialog box allows the user to select a font by its size, color, and style. Once the user makes selections in the Font dialog box, the following properties contain information about the user's selections.

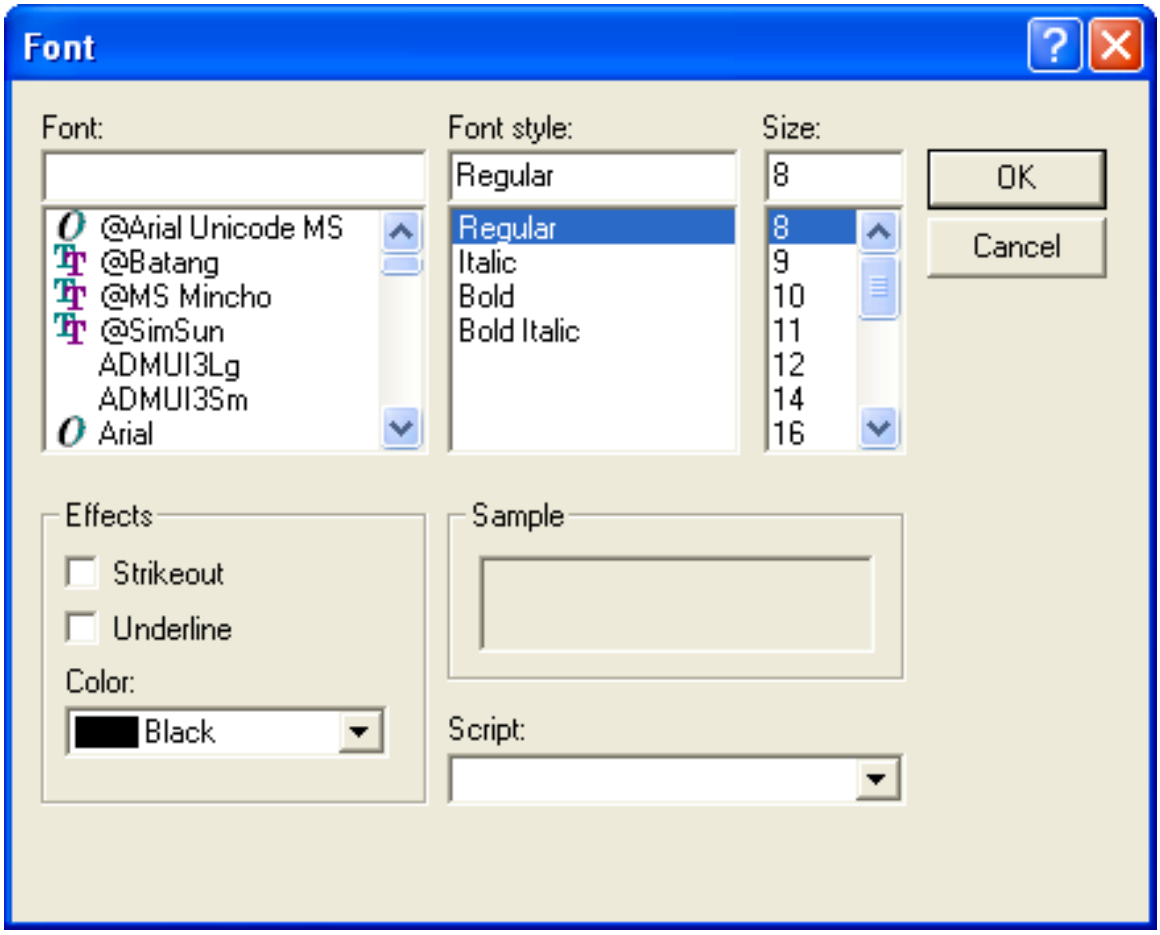


Figure 6

19.3.1 Properties

Property:	Determines:
Color	The selected color. To use this property, you must first set the Flags property to cdlCFEffects.
FontBold	Returns whether the bold checkbox was selected.
FontItalic	Returns whether the italic checkbox was selected.
FontStrikethru	Returns whether the strikethrough checkbox was selected.
FontUnderline	Returns whether the underline checkbox was selected.
FontName	Returns the selected font name.
FontSize	Returns the selected font size.

19.3.2 Before displaying the dialog

To display the dialog, you must first set the Flags property either cdlCFScreenFonts or cdlCFPrinterFonts (or cdlCFBoth, if you intend to let the user choose between both screen fonts and printer fonts).

19.3.3 Code example

```
    ' *** This example require a textbox called ''txtText'' to execute properly.
***
    On Error Resume Next

    ' Clear errors
    Err.Clear

    ' Use this common dialog control throughout the procedure
    With CommonDialog1

        ' Raises an error when the user press cancel
        .CancelError = True

        ' Show printer and screen fonts, as well as a font color chooser.
        .Flags = cdlCFBoth Or cdlCFEffects

        ' Display the font dialog box.
        .ShowFont

    End With

    ' Ignore this if the user has canceled the dialog
    If Err <> cdlCancel Then

        ' Set the textbox's font properties according to the users selections.
        With txtText
            .Font.Name = CommonDialog1.FontName
            .Font.Size = CommonDialog1.FontSize
            .Font.Bold = CommonDialog1.FontBold
            .Font.Italic = CommonDialog1.FontItalic
            .Font.Underline = CommonDialog1.FontUnderline
            .FontStrikethru = CommonDialog1.FontStrikethru
            .ForeColor = CommonDialog1.Color
        End With

    End If
```

19.4 Print dialog

The print dialog box allows the user to select how output should be printed. Options and properties accessible to the user includes the amount of copies to make, print quality, the range of pages to print, ect. It also allows the user to choose another printer, as well as configure different settings regarding the current printer.

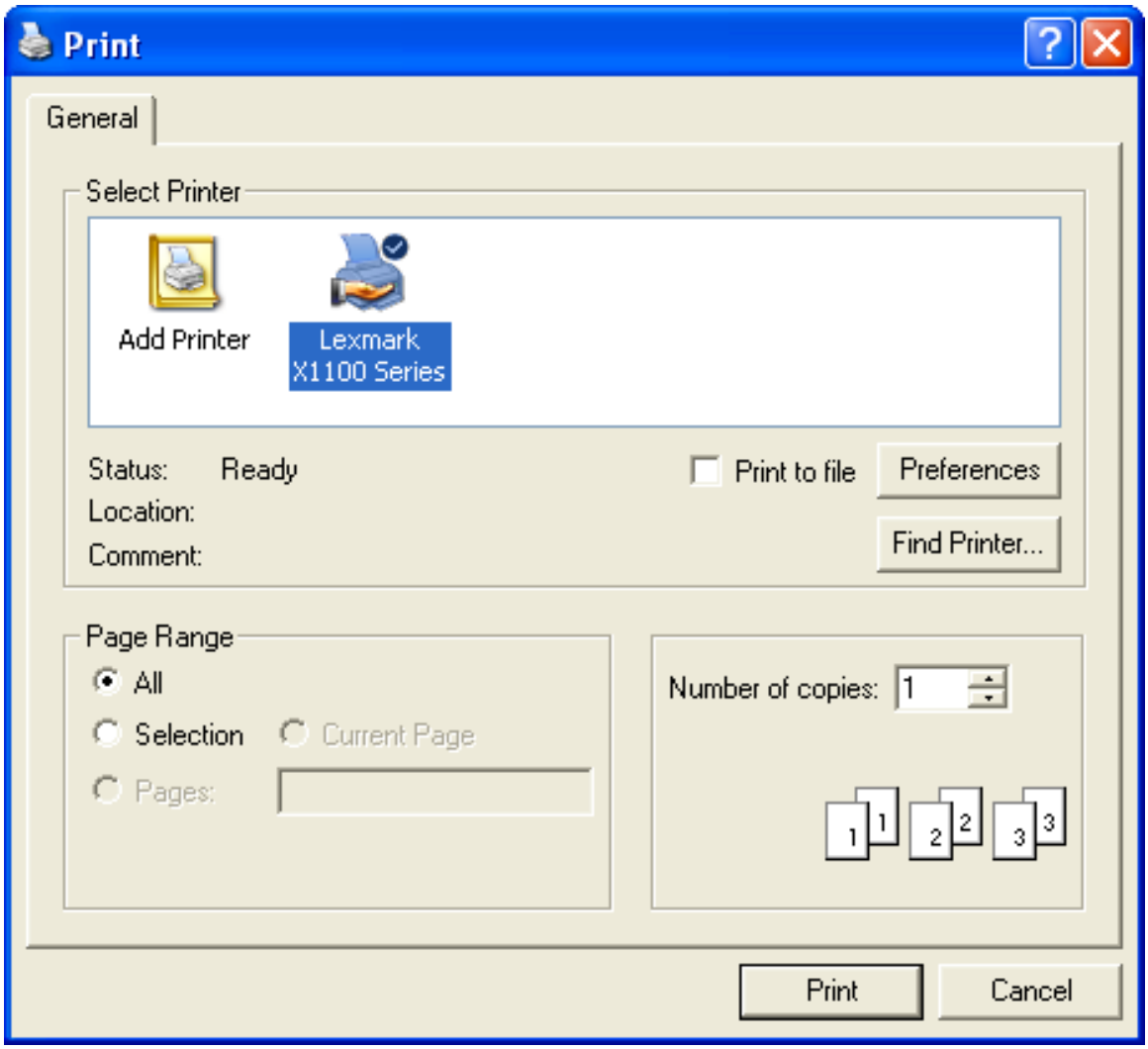


Figure 7

19.4.1 Properties

Property:	**Determines:**
Copies	The amount of copies to print.
FromPage	The starting page of the print range.
ToPage	The ending page of the print range.
hDC	The device context for the selected printer.

Property:	Determines:
Orientation	The page's orientation (as in portrait or landscape).

19.4.2 Before displaying the dialog

Before showing the dialog, feel free to set the appropriate print dialog properties to set the default values to use.

19.4.3 Code example

```
    On Error Resume Next

    ' Variables that holds information regarding the print setup.
    Dim BeginPage As Long, EndPage As Long, NumCopies As Long, Orientation As
Long, Tell As Long

    ' Clear errors
    Err.Clear

    ' Use this common dialog control throughout the procedure
    With CommonDialog1

        ' Raises an error when the user press cancel
        .CancelError = True

        ' Display the printer dialog box.
        .ShowPrinter

        ' Ignore this if the user has canceled the dialog
        If Err <> cdlCancel Then

            ' Retrieve the printer settings the user has chosen.
            BeginPage = CommonDialog1.FromPage
            EndPage = CommonDialog1.ToPage
            NumCopies = CommonDialog1.Copies
            Orientation = CommonDialog1.Orientation

            ' Start printing
            For i = 1 To NumCopies
                ' Put code here to send data to your printer.
            Next

        End If

    End With
```

20 Databases

Database management as a subject in its own right is far too big to be dealt with in this book and would distract from the principal subject which is programming in VB Classic. Nonetheless we can explain the basics of connecting to databases, retrieving information from them and so on.

The usual way to connect to a database from VB is to use ADO (ActiveX Data Objects).

Connecting to databases usually happens with a connection string. Here are some example connection strings to connect to some common database types

A

ACCESS

ODBC

Standard Security

```
Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\mydatabase.mdb;Uid=Admin;Pwd=;
```

Workgroup

```
Driver={Microsoft Access Driver
(*.mdb)};Dbq=C:\mydatabase.mdb;SystemDB=C:\mydatabase.mdw;
```

Exclusive

```
Driver={Microsoft Access Driver
(*.mdb)};Dbq=C:\mydatabase.mdb;Exclusive=1;Uid=admin;Pwd=;
```

Enable admin statements To enable certain programatically admin functions such as CREATE USER, CREATE GROUP, ADD USER, GRANT, REVOKE and DEFAULTS (when making CREATE TABLE statements) use this connection string.

```
Driver={Microsoft Access Driver
(*.mdb)};Dbq=C:\mydatabase.mdb;Uid=Admin;Pwd=;ExtendedAnsiSQL=1;
```

Specifying locale identifier Use this to specify the locale identifier which can help with non-US formatted dates.

```
Driver={Microsoft Access Driver (*.mdb)};Dbq=C:\mydatabase.mdb;Locale
Identifier=2057;Uid=Admin;Pwd=;
```

The above example uses the en-gb locale identifier (2057).

20.0.4 OLE DB, OleDbConnection (.NET)

Standard security

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\mydatabase.mdb;User
Id=admin;Password=;
```

With database password This is the connection string to use when you have an access database protected with a password using the Set Database Password function in Access.

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\mydatabase.mdb;Jet
OLEDB:Database Password=MyDbPassword;
```

Workgroup (system database)

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\mydatabase.mdb;Jet OLEDB:System
Database=system.mdw;
```

Workgroup (system database) specifying username and password

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\mydatabase.mdb;Jet OLEDB:System
Database=system.mdw;User ID=myUsername;Password=myPassword;
```

DataDirectory functionality

```
Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=|DataDirectory|\myDatabase.mdb;User Id=admin;Password=;
```

ACCESS 2007 Standard security

```
Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:\myFolder\myAccess2007file.accdb;Persist Security Info=False;
```

With database password

```
Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=C:\myFolder\myAccess2007file.accdb;Jet OLEDB:Database
Password=MyDbPassword;
```

Datadirectory functionality

```
Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=|DataDirectory|\myAccess2007file.accdb;Persist Security Info=False;
```

E

Excel Files

```
ODBC
```

Driver={Microsoft Excel Driver (*.xls)};DriverId=790;Dbq=C:\MyExcel.xls;DefaultDir=c:\mypath;

Remark:

```
SQL syntax "SELECT * FROM [sheet1$]". -- i.e. excel worksheet name followed by
a "$" and wrapped in "[" "]" brackets.
```

OLEDB

```
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\MyExcel.xls;Extended
Properties="Excel 8.0;HDR=Yes;IMEX=1";
```

Important note!

The quote " in the string needs to be escaped using your language specific escape syntax.

```
c#, c++   \"
VB6, VBScript   ""
xml (web.config etc)   "
or maybe use a single quote '.
```

"HDR=Yes;" indicates that the first row contains columnnames, not data. "HDR=No;" indicates the opposite.

"IMEX=1;" tells the driver to always read "intermixed" (numbers, dates, strings etc) data columns as text. Note that this option might affect Excel worksheet write access negative.

```
SQL syntax "SELECT * FROM [sheet1$]". -- i.e. Excel worksheet name followed by
a "$" and wrapped in "[" "]" brackets.
```

Check out the [HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\Engines\Excel] located registry REG_DWORD "TypeGuessRows". That's the key to not letting Excel use only the first 8 rows to guess the columns data type. Set this value to 0 to scan all rows. This might hurt performance. If the Excel workbook is protected by a password, you cannot open it for data access, even by supplying the correct password with your connection string. If you try, you receive the following error message: "Could not decrypt file." **Excel 2007 Files**

```
Provider=Microsoft.ACE.OLEDB.12.0;Data
Source=c:\myFolder\myExcel2007file.xlsx;Extended Properties="Excel 12.0
Xml;HDR=YES";
```

21 Windows API

APIs, short for "application programming interface", allows you to access a wide array of functions and methods exposed by the operating system or other applications (DLLs), performing operations not normally implemented by the standard virtual machine that Visual Basic provides. This includes operations such as shutting down the computer or listing all the current running processes.

Although VB6 compiler can't create true DLLs, it has ability to call existing DLLs (not necessarily just Windows API of course). Declaration of such function contains extra keyword **declare** and **lib**, plus optionally can be defined an alias if the real name of function differs from name of dll function **alias**. For example it can be:

```
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" (pDest As
 Any, pSource As Any, ByVal ByteLen As Long)
```

For datatype **Any** accepts compiler any parameter provided. For effective using of API calls are important operators **VarPtr** which returns pointer at variable specified and **AddressOf** which is important for callback functions. Beginners often do mistake on missing or badly specified ByRef/Byval in declaration. Simply said, **ByVal** means that on stack will be pushed value and **ByRef** (or missing specification of calling convention) means that on stack will be pushed pointer at variable. In practice parameter type Long (4 bytes) will be called ByVal, but String or complex data types will be called ByRef.

Unfortunately, VB6 doesn't support C-like memory handling. To deal with this problem VB programmers uses function CopyMemory to copy memory like this:

```
Dim a As Long
Dim b() As Byte
ReDim b(3)
CopyMemory ByVal varptr(a), ByVal varptr(b), 4
```

which is relatively safe. Some of programmers at the higher level of knowledge can modify pointers at objects as well. Although most of API functions can be called this way, there is not possible to call entry points of dlls dynamically (LoadLibrary-GetEntryPoint-FreeLibrary) for general case. Much clearer way how to deal with such tasks is to move code of this kind into separated dll written in C++ and call this dll.

21.1 Declaration

An example of a declaration is the GetTickCount function which returns the amount of milliseconds that have elapsed since Windows was started. To use it, insert the following into a standard module in your project:

```
Public Declare Function GetTickCount Lib "kernel32.dll" () As Long
```

To access it, simply call it like you would call any normal function. Note, however, that this is a *Public* declaration and is limited to modules. If you don't need it to be accessible in your entire project, use the *Private* declaration and insert it in the class module or form directly:

```
Private Declare Function GetTickCount Lib "kernel32.dll" () As Long
```

21.2 Resources

For a beginner it is normally difficult to create a declare statement since it requires knowing how to map C datatypes to Visual basic datatypes. There is a built in program that comes along with VB6 called API Text Viewer which lists all commonly used API functions, contants and types. A VB programmer can just find the required function from said program and insert into the VB module.

22 Subclassing

Subclassing can be useful when you want to add functionality that is not directly supported by Visual Basic. To explain how this works we must first go through a little background theory.

All *windows* in Windows (for example a form, a button, a listbox, etc.) have a function that the operating system or other programs can call to communicate with the program. Windows can, for example, send *messages* about *events* such as the mouse pointer moving over the window, a key being pressed when the window has the focus and much more. Programs can also send messages that ask for information about the window; for example, the *EM_GETLINECOUNT* message asks a textbox to send back the number of lines in the text it holds. One can also define one's own functions.

To call these special functions you can use *PostMessage*, *SendMessage* or *CallWindowProc* (the last only if you know the address of the function).

Usually such a procedure looks something like this:

```
Public Function WindowProc(ByVal hwnd As Long, _
                           ByVal uMsg As Long, _
                           ByVal wParam As Long, _
                           ByVal lParam As Long) As Long

  ' Your code here

  Select Case uMsg
    Case 0
      ' React to message 0
    Case 1
      ' React to message 1
  End Select

  WindowProc = 0 ' Return a value to the caller

End Function
```

In this function *hwnd* is the *handle* of the *window* that the caller has tried to contact; *uMsg* is the *message identifier* which says what the call is about; *wParam* and *lParam* are used for whatever purpose the caller and window agree on. The handle, hwnd, is not an address but is used by Windows to find the address.

If, for example, we want to set the text that appears on the title bar of a form we can use the following code:

```
Private Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
        (ByVal hwnd As Long, _
         ByVal wMsg As Long, _
         ByVal wParam As Integer, _
         ByVal lParam As Any) As Long
```

```
    Private Const WM_SETTEXT = &HC

  Private Sub Form_Load()
    SendMessage Me.hwnd, WM_SETTEXT, 0&, "This is a test"
  End Sub

The receiver will get this message via its window function, which will look
 something like this:

  Public Function WindowProc(ByVal hwnd As Long, _
                             ByVal uMsg As Long, _
                             ByVal wParam As Long, _
                             ByVal lParam As Long) As Long

    ' hwnd is now equal to Me.hwnd,
    ' uMsg is WM_SETTEXT,
    ' wParam is 0
    ' lParam is the address of the text: "This is a test"

    ' It doesn't actually look like this of course, but this gives
    ' a good enough impression of what happens under the surface
    Select Case uMsg
           Case WM_SETTEXT
        Me.Caption = lParam
      'Case ...
        '... many more here
    End Select
  End Function
```

22.1 Why Subclass

What is the point of *subclassing*?

Using this technique we can completely replace a program's own window function with our own. Then we can respond to the messages in ways that Visual Basic doesn't, we can decide to send the messages further to the original window function or not as we like, modifying them on the way in any way we please.

22.2 Example

To specify that our window function is to be used we use the *API* call *SetWindowLong*. Study the following example and put it in a base module:

```
  Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" _
          (ByVal hwnd As Long, _
           ByVal nIndex As Long, _
           ByVal dwNewLong As Long) As Long

  Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" _
          (ByVal lpPrevWndFunc As Long, _
           ByVal hwnd As Long, _
           ByVal Msg As Long, _
           ByVal wParam As Long, _
           ByVal lParam As Long) As Long

  Declare Function SetClipboardViewer Lib "user32" _
          (ByVal hwnd As Long) As Long
```

```
Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
        (ByVal hwnd As Long, _
         ByVal wMsg As Long, _
         ByVal wParam As Integer, _
         ByVal lParam As Any) As Long

Public Const WM_SETTEXT = &HC
Public Const GWL_WNDPROC = (-4)

Private PrevProc As Long ' The address of the original window function

Public Sub SubclassForm(F As Form)
  ' AddressOf WindowProc = finds the address of a function
  PrevProc = SetWindowLong(F.hwnd, GWL_WNDPROC, AddressOf WindowProc)
End Sub

Public Sub UnSubclassForm(F As Form)
  ' It is _very_ important that we restore the original window function,
  ' because VB will crash if we don't.
  SetWindowLong F.hwnd, GWL_WNDPROC, PrevProc
End Sub

Public Function WindowProc(ByVal hwnd As Long, _
                           ByVal uMsg As Long, _
                           ByVal wParam As Long, _
                           ByVal lParam As Long) As Long

  Dim sTemp As String

  If uMsg = WM_SETTEXT Then
    ' Don't let the text get through, replace it with our own. Also, because
all
    ' strings in VB are of the format UTF-16 (Unicode) and the receiving
method
    ' expects a zero-terminated ASCII-string, it is necessary to convert it
before
    ' passing it further down the chain.
    sTemp = StrConv("Subclassing" & Chr(0), vbFromUnicode)
    lParam = StrPtr(sTemp) ' get the address of our text
  End If

  ' Call the original function
  WindowProc = CallWindowProc(PrevProc, hwnd, uMsg, wParam, lParam)
End Function
```

Add a form with a button *cmdTest* and add this code:

```
Private Sub cmdTest_Click()
  SendMessage Me.hwnd, WM_SETTEXT, 0&, "This is a test"
End Sub

Private Sub Form_Load()
  ' Start subclassing
  SubclassForm Me
End Sub

Private Sub Form_Unload(Cancel As Integer)
  ' WARNING: If you stop the project (for example with the stop button)
without calling this,
  ' your program, as well as the VB IDE, will most likely crash.
  UnSubclassForm Me
End Sub
```

When you click the *cmdTest* button, you'll see that the text that appears is not "This is a test", but "Subclassing".

23 External Processes

It is often the case that a program already exists to do some job so rather than rewrite the program in Visual Basic it is more efficient to run the existing program from Visual Basic. This can work very well for command line programs or scripts written in any language.

The general scheme is to create a string containing the command line, call a function to run it and then wait for the command to complete. There are many refinements that can be made including all sorts of ways of sending data to the new process, reading results from it, pausing it, setting its priority and so on.

23.1 The Shell Function

The simplest way of running an external program is to do something like this:

```
Shell "cmd /c dir %temp%"
```

Shell is a built in Visual Basic function that executes a command line and returns a *handle* to that process. *Shell* takes an optional argument that controls the *window style* of the new process (maximized, normal, hidden, etc.).

Unfortunately running another program like this doesn't give you much control over what is happening. In particular, there is no obvious way to find out if the program has completed its work.

Luckily *Shell* actually does slightly more than launch the program. It also returns the *process id*. This is the number displayed by *Task Managers* and can be used to check the status of the *process*. Unfortunately, as is so often the case with Visual Basic and Windows, there is no built in function nor even a single API[1] function that can use it directly. Therefore, we must write a little more code. First, we must declare a few API[2] functions and constants:

```
  Option Explicit

  Const SYNCHRONIZE = &H100000
  Const INFINITE = &HFFFF     'Wait forever
  Const WAIT_OBJECT_0 = 0     'The state of the specified object is signaled.
  Const WAIT_TIMEOUT = &H102 'The time-out interval elapsed, and the object's
 state is nonsignaled.

  Private Declare Function OpenProcess Lib "kernel32" _
          (ByVal dwDesiredAccess As Long, _
           ByVal bInheritHandle As Long, _
           ByVal dwProcessId As Long) _
```

1 Chapter 20.0.4 on page 106
2 Chapter 20.0.4 on page 106

```
        As Long
Private Declare Function WaitForSingleObject Lib "kernel32" _
        (ByVal hHandle As Long, _
         ByVal dwMilliseconds As Long) _
        As Long
Private Declare Function CloseHandle Lib "kernel32" _
        (ByVal hObject As Long) _
        As Long

Dim ProcessID As Long
Dim hProcess As Long
```

Here is an example of a function that executes a command and waits for it to complete:

```
Function ShellWait(CommandLine As String, _
                   TimeOut As Long, _
                   WindowState As VbAppWinStyle) As Boolean

  Dim ProcessID As Long
  Dim hProcess As Long

  ProcessID = Shell(CommandLine,WindowState)
  If ProcessID <> 0 Then
    'non-zero (True) so Shell worked
    ' Get a process handle for the PID (Wait takes a handle)
    hProcess = OpenProcess(SYNCHRONIZE, False, ProcessID)
    If hProcess <> 0 Then
      ' Got process handle
      ' Wait until process finishes before going on
      If WaitForSingleObject(hProcess, TimeOut) = WAIT_OBJECT_0 Then
        ShellWait = True
      Else
        ShellWait = False
      End If
    Else
      'Failed to get process handle.
      'Perhaps the process terminated very quickly
      'or it might not really have executed at all even though Windows
      ' started a process.
      ShellWait = False
    End If
  Else
    ' PID zero (False) so Shell failed
    ShellWait = False
  End If
End Function
```

Call it like this:

```
If ShellWait("calc.exe") then
  MsgBox "Success :-)"
Else
  MsgBox "Failure :-("
End If
```

23.2 Exercises

- Modify the ShellWait example function so that the caller can distinguish between failure to launch the process and timeout.
- Suggest a reason why this might be a good idea.

24 Object Oriented Programming

We will now move into a more advanced aspect of VB - OOP. But don't worry, Object Oriented Programming is quite simple, in fact it is probably simpler for those who have never programmed before than for those with long experience of traditional Fortran/Basic/Pascal (pick your favourite imperative language).

You should be familiar with VB's event-driven programming style by now. However I'll explain it again. When you're doing general VB programming, your thoughts should go in this pattern: "If the user does this, what should happen? How would I make it happen?" Then you would write the program to fit the answers to these questions. That is event-driven programming. You program according to what events the user would trigger. Dragging a picture, clicking a button, and typing a word are all events.

You, being the brave coder that you are, would ask: "WHY do I HAVE to think like that??"

Well here's an alternative way for you to think: Object Oriented Programming(OOP). In the world of OOP, you break a problem down into small parts and solve them individually. If you are to program in an object oriented style, you would think of every variable or functions as a property of an object, and everything would seem like an object to you. OOP is hard to explain so you'll have to experience it in the following examples.

Imagine that you have a bottle of juice. What properties does it have? What events does it have?(What can it do?) Here's a list I could think of:

```
Private Colour As String  'What colour?
Private Fruit As String   'What kind of fruit?
Private Volume As Single  'How much?
Public Sub MakeJuice(c As String, f As String, v As Single)  'Make some
  Colour = c
  Fruit = f
  Volume = v
End Sub

Public Sub Evaporate()   'Well, that's the only thing I could think of
  Volume = Volume - 10
End Sub
```

Behold. That's our first class. Don't worry about anything right now. I will go over those later. Anyway, let's assume that this class is named, "Juice". Think of it as a general description of a bottle of juice. Now we can create an actual bottle of apple juice:

```
Private AppleJuice As Juice  'Define
Set AppleJuice = New Juice   'Create a new instance of Juice
AppleJuice.MakeJuice "brown", "apple", 30
```

Watch:

```
AppleJuice.Evaporate   '20 units left
```

```
AppleJuice.Evaporate   '10 left
AppleJuice.Evaporate   'Yes!!
```

"Wait", you interrupted my rant once again, "but why do I have to use OOP? Just because you hate apple juice?"

OOP is good for large-scale programming - As your code grows large, there are going to be more and more procedures/functions to your program, and your code is going to be so cluttered that one more look at it, you'd scream. That is why I'm teaching you OOP! (So instead of looking at hundreds of scattered procedures/functions, you look at organized Objects.)

24.1 Types

You can think of a type as a simple form of class. Types can only hold variables, not procedures/functions.

An important distinction between user defined types ('UDT') and 'classes' in 'VB6' is that 'UDTs' are *value types* while 'classes' are *reference types*. This means that when you store an object ('instance' of a 'class') in a variable what actually happens is that a pointer to the existing object is stored but when you store a 'UDT' instance in a variable a *copy* of the instance is stored. For instance imagine that you have defined a class, 'Class1', and a UDT, 'Type1':

```
Dim ac As Class1
Dim bc As Class1
Dim at As Type1
Dim bt As Type1
```

Now assign ac to bc:

```
Set bc = ac
```

and at to bt

```
bt = at
```

Now make a change to one of the properties of bc and look at the corresponding property in ac. You should see that when you change bc that ac also changes. This is because ac and bc are really pointers to the same block of memory. If you do the same exercise with bt and at you should see that changing properties of bt does not affect at. This is because at and bt are values not references and occupy distinct locations in memory.

This is important when storing objects in 'Collections'. If you add an object to a collection you can later change the values of its properties and the object in the collection will also seem to change. in fact it is the same object because what is stored in the collection is the reference ('pointer') to the object. However if you store a 'UDT' in a collection what is stored is a *copy* of the *values* not a reference so changing the original UDT will not affect the content of the Collection.

24.2 Classes

Classes are a combination of subroutines and types. By that I mean that when you write a class it looks very much like a type[1] but is also contains subroutines and functions.

Each class in VB6 is a distinct file, there can only be one class in a file and the class cannot be spread out in several files; this is a feature of VB6 not a requirement of object oriented programming.

A class looks very much like an ordinary module but has the file extension *.cls* instead of *.bas*

24.3 Inheritance

There are two types of inheritance: implementation and interface. However Visual Basic Classic provides only Interface inheritance. Implementation inheritance can be simulated by a combination of interface inheritance and delegation.

One of the common ways to introduce inheritance is to show a program that creates dog and cat objects or car and train objects.

For example in Visual Basic we can declare two classes dog and cat like this:

24.3.1 Class cDog

```
Option Explicit

Public Sub Sound()
  Debug.Print "Woof"
End Sub
```

24.3.2 Class cCat

```
Option Explicit

Public Sub Sound()
  Debug.Print "Meow"
End Sub
```

24.3.3 module main

```
Public Sub main()
  Dim oDog as cDog
  Dim oCat as cCat

  Set oDog = New cDog
```

1 Chapter 12 on page 72

```
    Set oCat = New cCat

    oDog.Sound
    oCat.Sound

  End Sub
```

When you run this program it will print:

```
    Woof
    Meow
```

Now this is all very well until you decide that you would like an array of pets:

```
Dim aPets(1 to 10) as ????
```

Unless you declare the *aPets* array as *variant* you have to give it a type. One way is to create a class called *cPet* and use it instead of cDog and cCat:

24.3.4 Class cPet

```
  Option Explicit

  Public IsDog as Boolean

  Public Sub Sound()
    If IsDog Then
      Debug.Print "Woof"
    Else
      Debug.Print "Meow"
    End If

  End Sub
```

Now our *main* routine might look like this:

```
  Option Explicit

  Dim aPet(1 to 10) as cPet

  Public Sub main()

    Dim lPet as Long
    For lPet = 1 to 10
      Set aPet(lPet) = New cPet
      If lPet<=5 then
        aPet(lPet).IsDog = True
      Else
        aPet(lPet).IsDog = False
      End If
    Next lPet

    ' Now find out what noise they make.
    For lPet = 1 to 10
      aPet(lPet).Sound
    Next lPet

  End Sub
```

This should print:

```
Woof
Woof
Woof
Woof
Woof
Meow
Meow
Meow
Meow
Meow
```

Look reasonable? For the requirements so far revealed this works fine. But what happens when you discover that there are things that cats do that have no analogue in dogs? What about other kinds of pet? For example, you want your cat to *purr*. You can have a method called *purr* but where will you put it? It surely doesn't belong in cPet because dogs can't purr and neither can guinea pigs or fish.

The thing to do is to separate out those features that are common to all *pets* and create a class that deals only with those, we can rewrite *cPet* for this purpose. Now we can go back to our original *cDog* and *cCat* classes and modify them to *inherit* the 'interface*from*cPet.

24.3.5 Class cPet rewritten

```
Option Explicit

Public Sub Sound()
End Sub
```

Notice that the *Sound* method is empty. this is because it exists only to define the interface. The interface is like the layout of the pins on a plug or socket; any plug that has the same layout, size and shape will plug in to a socket implementing the same interface. What goes on inside the thing you plug in is a separate issue; vacuum cleaners and television sets both use the same mains plug.

To make this work we must now modify cDog and cCat:

24.3.6 Class cDog

```
Option Explicit
Implements cPet

Private Sub cPet_Sound()
  Debug.Print "Woof"
End Sub
```

24.3.7 Class cCat

```
Option Explicit
Implements cPet
```

```
Private Sub cPet_Sound()
  Debug.Print "Meow"
End Sub

Public Sub Purr()
  Debug.Print "Purr"
End Sub
```

24.4 Collections

In the pet examples above an array of cPet objects is used to hold the pets. This works well if you know exactly how many pets you have. But what happens if you adopt another stray puppy or your cat has kittens?

One solution is to use a Collection object instead of an array. A collection object looks a little like an array but lets you add and remove items without worrying about the declared size, it just expands and contracts automatically.

24.5 Iterators

25 Effective Programming

When programming effectively in any computer language, whether it is VB or C++ for example, there should be consistency to your style, it should be organized, and should aim to be as efficient as possible in terms of speed of execution and use of resources (such as memory or network traffic). With established programming techniques, errors can be reduced to a minimum and be more easily recognized, and it will make the job of the programmer much easier and more enjoyable.

There are many different aspects to writing reliable programs. For a short program used interactively and only by the author it can be reasonable to break all the rules in the service of quickly getting the answer. However, if that little programs grows into a large program you will end up wishing that you had started on the right path. Each programming language has its strengths and weaknesses and a technique that assists in writing good programs in one might be unnecessary, impossible or counterproductive in another; what is presented here applies to VB6 in particular but much of it is standard stuff that applies or is enforced in Pascal, C, Java and other similar imperative languages.

25.1 General Guidelines

These suggestions will be described in greater detail further below. None of these can be called rules and some are controversial, you have to make up your own mind based on the costs and benefits

- Write comments that explain why you do what you do. If the code or the problem being solved is especially complex you should also explain why you chose the method you did instead of some other more obvious method.
- Indent your code. This can make reading the code for others easier and makes it easy to spot where statements have not been closed off properly. This is especially important in multiply nested statements.
- Declare all variables, enforce this by placing *Option Explicit* at the top of every code module,
- Use meaningful variable and sub routine names. The variable FileHandle means a lot more to us humans than X. Also avoid the tendency of abbreviating names as this can also make it hard to read code. Don't use FilHan where FileHandle would be clearer,
- In the argument list of functions and subs declare all arguments as *ByRef*. This forces the compiler to check the datatypes of the variables you pass in,
- Declare variables, subs, and functions in the smallest possible scope: prefer Private over Friend and Friend over Public.

- Have as few variables as possible declared Public in .bas modules; such variables are public to the whole component or program.
- Group related functions and subs together in a module, create a new module for unrelated routines,
- If a group of variables and procedures are closely related consider creating a class to *encapsulate* them together,
- Include *assertions* in the code to ensure that routines are given correct data and return correct data,
- Write and execute *tests*,
- Make the program work first and work fast afterwards
- Where a variable can hold a limited range of discrete values that are known at compile time use an 'enumerated type,
- Break large programs into separate components (DLLs or class libraries) so that you can reduce the visibility of data and routines, to just those other pieces of code that need to use them,
- Use a simple prefix notation to show the type of variables and the scope of routines.

25.2 Declaring variables

Earlier in this book, you may have been taught to declare variables with a simple Dim statement, or not at all. Declaring variables on different levels is a crucial skill. Think of your program as three branches: The module (open to all forms), individual forms, and the sub programs themselves. If you declare a variable in your module, the variable will retain its value through all forms. A dim statement will work, but it is tradition to use "Public" in its place. For example:

```
Public X as Integer
```

Declaring something at the top of your form code will make it private to that form, therefore, if you have X=10 in one form, and X=20 in another, they will not interfere. If the variable was declared public, then there would be interactions between the values. To declare something in your form, it is traditional to use "Private".

```
Private X as Integer
```

And finally, there are the subprograms. Dimensioning variables only to a subprogram is highly effective, this way you can use default variables (such as sum for sums) in all subs without the need to worry about one value changing because of another section of code. There is, however, a twist. Dim, what you are accustomed to, will not retain the value of the variable after the sub is done. So after rerunning the sub, all the local variables in the sub will be reset. To get around this, a "Static" may be used.

```
Static X as Integer
```

Some of you may want to take the easy way out, and just use Public on everything. However, it is best to declare something on the smallest level possible. Generally arguments are the best way to send variables from one sub to the other. This is because arguments make it far easier to track exactly where variables are being changed in case of a logic error, and almost limits the amount of damage a bad section of code can do. Declaring variables is, again, useful when using default variables. Common default variables are:

```
I for loops
J for loops
Sum(self explanatory)
X for anything
```

So, rather then making variables I and II or Sum1,Sum2, you can see why keeping variables local is a useful skill.

25.3 Comments

Every programmer I know dislikes writing comments. I don't mean just illiterate script-kiddies writing some brain dead Visual Basic Script to delete all the files from some poor grandmother's PC, I mean them and everyone up to people with multiple Ph.D.s and honorary doctorates.

So if you find it difficult to convince yourself that comments are a good idea you are in good company.

Unfortunately this isn't a case of you and I being as good as them but of them being as bad as us. Good comments can be critical to the longevity of a program; if a maintenance programmer can't understand how your code was supposed to work he might have to rewrite it. If he does that he will also have to write more comments and more tests. He will almost certainly introduce more bugs and it will be your fault because you didn't have the courtesy to explain why your program was written the way it was. Maintenance programmers are often not part of the original team so they don't have any shared background to help them understand, they just have a bug report, the code and a deadline. If you don't write the comments you can be sure that no one will add them later.

Comments need to be written with the same care and attention as the code itself. Sloppily written comments that simply repeat what the code says are a waste of time, better to say nothing at all. The same goes for comments that contradict the code; what is the reader meant to believe, the code or the comment. If the comment contradicts the code someone might 'fix' the code only to find that it was actually the comment that was broken.

Here are some example comments:

```
Dim cbMenuCommandBar As Office.CommandBarButton 'command bar object
```

That comes from one of my own programs that I created by hacking at a template provided by someone else. Why he added the comment is beyond me, it adds nothing to the code which contains the word CommandBar twice anyway!

Here is another from a similar program:

```
Public WithEvents MenuHandler As CommandBarEvents 'command bar event handler
```

Same degree of pointlessness. Both examples are from programs that I use everyday.

Another from the same program which shows both a good comment and a pointless one:

```
Public Sub Remove(vntIndexKey As Variant)
  'used when removing an element from the collection
  'vntIndexKey contains either the Index or Key, which is why
  'it is declared as a Variant
  'Syntax: x.Remove(xyz)

  mCol.Remove vntIndexKey
End Sub
```

The first comment line simply repeats the information contained in the name, the last comment line tells us only what we can easily glean from the declaration. The middle two lines say something useful, they explain the otherwise puzzling use of the *Variant* data type. My recommendation is to delete the first and last comment lines, not because they are incorrect but because they are pointless and make it harder to see the comment that actually matters.

To summarize: good comments explain *why* not *what*. They tell the reader what the code cannot.

You can see what is happening by reading the code but it is very often hard or impossible to see why the code is written as it is.

Comments that paraphrase the code do have a place. If the algorithm is complex or sophisticated you might need to precis it in plain human language. For example the routines that implement an equation solver need to be accompanied by a description of the mathematical method employed, perhaps with references to textbooks. Each individual line of code might be perfectly clear but the overall plan might still be obscure; a precis in simple human language can make it plain.

If you make use of a feature of VB that you know is little used you might need to point out why it works to prevent well-meaning maintenance programmers cleaning it up.

If your code is solving a complex problem or is heavily optimized for speed it will need more and better comments than otherwise but even simple code needs comments that explain why it exists and outlines what it does. Very often it is better to put a narrative at the head of a file instead of comments on individual code lines. The reader can then read the summary instead of the code.

25.3.1 Summary

- Comments should add clarity and meaning,
- Keep the comments short unless the complexity of the code warrants a narrative description,
- Add comments to the head of each file to explain why it exists and how to use it,
- Comment each function, subroutine and property to explain any oddities such as the use of the Object or Variant data types.

- If a function has side effects explain what they are,
- If the routine is only applicable to a certain range of inputs then say so and indicate what happens if the caller supplies something unexpected.

25.3.2 Exercises

- Take a piece of code written by someone else and try to understand how it works without reading the comments.
- Try to find some code that doesn't need any comments. Explain why it doesn't need them. Does such a thing exist?
- Search the web, your own code or a colleague's code to find examples of good comments.
- Put yourself in the position of a maintenance programmer called in to fix a bug in a difficult piece of your own code. Add comments or rewrite the existing comments to make the job easier.

25.4 Avoid Defensive Programming, Fail Fast Instead

By defensive programming I mean the habit of writing code that attempts to compensate for some failure in the data, of writing code that assumes that callers might provide data that doesn't conform to the contract between caller and subroutine and that the subroutine must somehow cope with it.

It is common to see properties written like this:

```
Public Property Let MaxSlots(RHS as Long)
  mlMaxSlots = RHS
End Property

Public Property Get MaxSlots() as Long
  if mlMaxSlots = 0 then
    mlMaxSlots = 10
  End If
  MaxSlots = mlMaxSlots
End Property
```

The reason it is written like this is probably that the programmer of the class in which it sits is afraid that the author of the code that uses it will forget to initialize the object properly. So he or she provides a default value.

The problem is that the programmer of the MaxSlots property has no way of knowing how many slots that the client code will need. If the client code doesn't set MaxSlots it will probably either fail or, worse, misbehave. It is much better to write the code like this:

```
Public Property Let MaxSlots(RHS as Long)
  mlMaxSlots = RHS
End Property

Public Property Get MaxSlots() as Long
  if mlMaxSlots = 0 then
    Err.Raise ErrorUninitialized, "MaxSlots.Get", "Property not initialized"
  End If
  MaxSlots = mlMaxSlots
End Property
```

Now when client code calls MaxSlots Get before calling MaxSlots Let an error will be raised. It now becomes the responsibility of the client code to fix the problem or pass on the error. In any case the failure will be noticed sooner than if we provide a default value.

Another way of viewing the distinction between *Defensive Programming* and *Fail Fast* is to see *Fail Fast* as strict implementation of a contract and *Defensive Programming* as forgiving. It can be, and is often, debated whether being forgiving is good always a good idea in human society but within a computer program it simply means that you don't trust all parts of the program to abide by the specification of the program. In this case you need to fix the specification not forgive transgressions of it.

25.4.1 Exercises

- Take a working non-trivial program and search the code for defensive programming,
- Rewrite the code to make it fail fast instead,
- Run the program again and see if it fails,
- Fix the client part of the program to eliminate the failure.

25.4.2 References

For a succinct article on this subject see `http://www.martinfowler.com/ieeeSoftware/failFast.pdf`.

25.5 Assertions and Design By Contract

An *assertion* is a statement that *asserts* that something is true. In VB6 you add assertions like this:

```
Debug.Assert 0 < x
```

If the statement is true then the program continues as if nothing happened but if it is not then the program will stop at that line. Unfortunately VB6 has a particularly weak form of assertions, they are only executed when the code is running in the debugger. This means that they have no effect at all in the compiled program. Don't let this stop you from using them, after all this is a feature that doesn't exist at all in many common programming languages.

If you really need the assertions to be tested in the compiled program you can do something like this:

```
If Not(0 < x) Then
  Debug.Assert False
  Err.Raise ErrorCodes.AssertionFailed
End If
```

Now the program will stop at the failure when in the IDE and raise an error when running compiled. If you plan to use this technique in more than a few places it would make sense to declare a subroutine to do it so as to reduce clutter:

```
Public Sub Assert(IsTrue as Boolean)
  If Not IsTrue Then
    Debug.Assert False
    Err.Raise ErrorCodes.AssertionFailed
  End If
End Sub
```

then instead of writing *Debug.Assert* you write:

```
Assert 0 < x
```

Assertions can be used to implement a form of design by contract[1]. Add assertions at the beginning of each routine that assert something about the values of the arguments to the routine and about the values of any relevant module or global variables. For instance a routine that takes a single integer argument that must be greater than zero would have an assertion of the same form as the one shown above. If it is called with a zero argument the program will halt on the line with the assertion. You can also add assertions at the exit of the routine that specify the allowed values of the return value or any side effects.

Assertions differ from explicit validation in that they do not raise errors or allow for the program to take action if the assertion fails. This is not necessarily a weakness of the assertion concept, it is central to the different ways that assertions and validation checks are used.

Assertions are used to do several things:

- They specify the contract that must be followed by the calling and called code,
- They assist in debugging by halting execution at the earliest point at which it is known that something is wrong. Correctly written assertions catch errors long before they cause the program to blow up.

Assertions generally assist in finding logic errors during the development of a program, validation, on the other hand, is usually intended to trap poor input either from a human being or other external unreliable source. Programs are generally written so that input that fails validation does not cause the program to fail, instead the validation error is reported to a higher authority and corrective action taken. If an assertion fails it normally means that two internal parts of the program failed to agree on the terms of the contract that both were expected to comply with. If the calling routine sends a negative number where a positive one is expected and that number is not provided by the user no amount of validation will allow the program to recover so raising an error is pointless. In languages such as C failed assertions cause the program to halt and emit a stack trace but VB simply stops when running in the IDE. In VB assertions have no effect in compiled code.

Combined with comprehensive tests, assertions are a great aid to writing correct programs. Assertions can also take the place of certain types of comments. Comments that describe the allowed range of values that an argument is allowed to have are better written as assertions because they are then explicit statements in the program that are actually checked. In VB you must exercise the program in the IDE to get the benefit of the assertions, this is a minor inconvenience.

1 http://en.wikipedia.org/wiki/design%20by%20contract

Assertions also help ensure that a program remains correct as new code is added and bugs are fixed. Imagine a subroutine that calculates the equivalent conductance of a radiator due to convection (don't worry if the physics is unfamilar):

```
 Public Function ConvectionConductance(Byval T1 as Double, Byval T2 as Double)
as Double
   ConvectionConductance = 100 * Area * (T2 - T1)^0.25
 End Sub
```

Now it so happens, if you know the physics, that conductance is always a non-negative number regardless of the relationship between the temperatures T1 and T2. This function, however, makes the assumption that T1 is always greater than, or equal to, T2. This assumption might be perfectly reasonable for program in question but it is a restriction nonetheless so it should be made part of the contract between this routine and its callers:

```
 Public Function ConvectionConductance(Byval T1 as Double, Byval T2 as Double)
as Double
   Debug.Assert T2 < T1
   ConvectionConductance = 100 * Area * (T2 - T1)^0.25
 End Sub
```

It is also a good a idea to assert something about the results of a routine:

```
 Public Function ConvectionConductance(Byval T1 as Double, Byval T2 as Double)
as Double
   Debug.Assert T2 <= T1
   ConvectionConductance = 100 * Area * (T2 - T1)^0.25
   Debug.Assert 0 <= ConvectionConductance
 End Sub
```

In this particular case it looks as if the assertion on the result is worthless because if the precondition[2] is satisfied it is obvious by inspection that the postcondition[3] must also be satisfied. In real life the code between the precondition[4] and postcondition[5] assertions is usually much more complicated and might include a number of calls to functions that are not under the control of the person who created the function. In such cases the postcondition[6] should be specified even if it appears to be a waste of time because it both guards against the introduction of bugs and informs other programmers of the contract that the function is supposed to comply with.

25.6 Tests

Tests vary from writing a program and then running it and looking casually at its behaviour to writing a full suite of automated tests first and then writing the program to comply.

Most of us work somewhere in between, usually nearer the first alternative than the second. Tests are frequently regarded as an extra cost but like quality control systems for physical

2 http://en.wikipedia.org/wiki/Precondition
3 http://en.wikipedia.org/wiki/Postcondition
4 http://en.wikipedia.org/wiki/Precondition
5 http://en.wikipedia.org/wiki/Postcondition
6 http://en.wikipedia.org/wiki/Postcondition

products the supposed cost of quality is often negative because of the increased quality of the product.

You can use tests to define the specification of a function or program by writing the tests first, this is one of the practices of the Extreme Programming method. Then write the program piece by piece until all of the tests pass. For most people this seems like a counsel of perfection and entirely impractical but a certain amount of it will pay off handsomely by helping to make sure that component parts work properly before integration.

A test is usually built as a separate program that uses some of the same source code as the deliverable program. Simply writing another program that can use component parts of the deliverable will often be enough to expose weaknesses in the design.

The smaller the component that is being tested the easier it will be to write a test, however if you test very small pieces you can waste a lot of time writing tests for things that can be easily checked by eye. Automated tests are probably best applied to those parts of the program that are small enough to be extracted from the real program without disruption yet large enough to have some complex behaviour. It's hard to be precise, better to do some testing than none and experience will show you where the effort is best expended in your particular program.

You can also make the tests part of the program itself. For instance, each class could have a test method that returns true the test passes and false otherwise. This has the virtue that every time you compile the real program you compile the tests as well so any changes in the program's interface that will cause the tests to fail are likely to be captured early. Because the tests are inside the program they can also test parts that are inaccessible to external test routines.

25.7 Scope, Visibility and Namespaces

25.8 Hungarian Notation

Hungarian notation is the name for the prefixes that many programmers add to variable names to denote scope and type. The reason for doing it is to increase the readability of the code by obviating the need to keep referring to the variable declarations in order to determine the type or scope of a variable or function.

Experienced Basic programmers have been familiar with a form of this notation for a very long time because Microsoft Basic's have used suffixes to indicate type (# means *Double*, & means *Long*, etc.).

The fine details of the Hungarian Notation used in any given program don't matter very much. The point is to be consistent so that other programmers reading your code will be able to quickly learn the conventions and abide by them. For this reason it is wise to not overdo the notation, if there are too many different prefixes people will forget what the rarely used ones mean and that defeats the purpose. It is better to use a generic prefix that will be remembered than a host of obscure ones that won't.

A recommendation for Hungarian Notation is described in greater detail in the Coding Standards[7] chapter.

25.9 Memory and Resource Leaks

You might think that because Visual Basic has no native memory allocation functions that memory leaks would never occur. Unfortunately this is not the case; there are several ways in which a Visual Basic program can leak memory and resources. For small utility programs memory leaks are not a serious problem in Visual Basic because the leak doesn't have a chance to get big enough to threaten other resource users before the program is shut down.

However, it is perfectly reasonable to create servers and daemons in Visual Basic and such programs run for a very long time so even a small leak can eventually bring the operating system to its knees.

In Visual Basic programs the most common cause of memory leaks is circular object references. This problem occurs when two objects have references to each other but no other references to either object exist.

Unfortunately the symptoms of a memory leak are hard to spot in a running program, you might only notice when the operating system starts complaining about a shortage of memory.

Here is an example problem that exhibits the problem:

```
'Class1
Public oOther as Class1

'module1
Public Sub main()
  xProblem
End Sub

Private Sub xProblem
  Dim oObject1 As Class1
  Dim oObject2 As Class1
  set oObject1 = New Class1
  set oObject2 = New Class1
  set oObject1.oOther = oObject2
  set oObject2.oOther = oObject1
End Sub
```

Class1 is a simple class with no methods and a single Public attribute. Not good programming practice for real programs but sufficient for the illustration. The xProblem subroutine simply creates two instances of Class1 (objects) and links them together. Notice that the oObject1 and oObject2 variables are local to xProblem. This means that when the subroutine completes the to variables will be discarded. When Visual Basic does this it decrements a counter in each object and if this counter goes to zero it executes the Class_Terminate method (if there is one) and then recovers the memory occupied by the object. Unfortunately, in this case the reference counter can never go to zero because each object refers to the other so even though no variable in the program refers to either of the objects they will

7 Chapter 31.8 on page 191

never be discarded. Any language that uses a simple reference counting scheme for cleaning up object memory will suffer from this problem. Traditional C and Pascal don't have the problem because they don't have garbage collectors at all. Lisp and its relatives generally use some variant of mark and sweep garbage collection which relieves the programmer of the problem at the expense of unpredictable changes in resource load.

To demonstrate that there really is a problem add *Initialize* and *Terminate* event handlers to *Class1* that simply print a message to the *Immediate Window*.

```
Private Sub Class_Initialize()
  Debug.Print "Initialize"
End Sub

Private Sub Class_Terminate()
  Debug.Print "Terminate"
End Sub
```

If the *xProblem* routine were working without a leak you would see an equal number of *Initialize* and *Terminate* messages.

25.9.1 Exercises

- Modify *xProblem* to ensure that both objects are disposed of when it exits (hint: setting a variable to *Nothing* reduces the reference count of the object it points to).

25.9.2 Avoiding and Dealing with Circular References

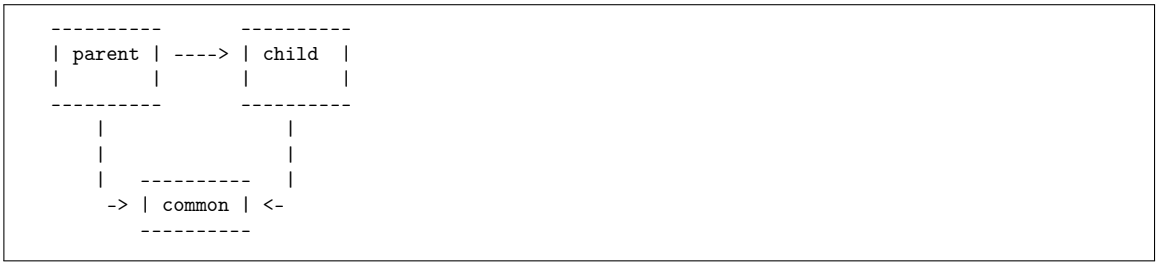
There are a number of techniques that can be used to avoid this problem beginning with the obvious one of simply never allowing circular references:

- Forbid circular references in your programing style guide,
- Explicitly clean up all references,
- Provide the functionality by another idiom.

In real programs forbidding circular references is not usually practical because it means giving up the use of such useful data structures as doubly linked lists.

A classic use of circular references parent-child relationships. In such relationships the *parent* is the *master* object and owns the *child* or *children*. The parent and its children share some common information and because the information is common to all of them it is most natural that it be owned and managed by the parent. When the parent goes out of scope the parent and all the children are supposed to be disposed of. Unfortunately this won't happen in Visual Basic unless you help the process along because in order to have access to the shared information the children must have a reference to the parent. This is a circular reference.

In this particular case you can usually avoid the child to parent reference completely by introducing a helper object. If you partition the parent's attributes into two sets: one which contains attributes that only the parent accesses and another that is used by both parent and children you can avoid the circularity by placing all those shared attributes in the helper object. Now both parent and have reference to the helper object and no child needs a reference to the parent.



Notice how all the arrows point away from the parent. This means that when our code releases the last reference to the parent that the reference count will go to zero and that the parent will be disposed of. This,in turn, releases the reference to the child. Now with both parent and child gone there are no references left to the *common* object so it will be disposed of as well. All the reference counting and disposal takes place automatically as part of Visual Basic's internal behaviour, no code needs to be written to make it happen, you just have to set up the structures correctly.

Note that the parent can have as many children as you like, held in a collection or array of object references for instance.

A common use for this sort of structure occurs when the child needs to combine some information about the parent with some of its own. For instance, if you are modelling some complicated machine and want each part to have a property showing its position. You would like to avoid making this a simple read write property because then you have to explicitly update that property on each object when the machine as a whole moves. Much better to make it a calculated property based on the parent position and some dimensional properties then when the parent is moved all the calculated properties will be correct without running any extra code. Another application is a property that returns a fully qualified path from root object to the child.

Here is a code example:

```
'cParent
Private moChildren as Collection
Private moCommon as cCommon

Private Sub Class_Initialize()
  Set moChildren = New Collection
  Set moCommon = New cCommon
End Sub

Public Function NewChild as cChild
  Set NewChild = New cChild
  Set NewChild.oCommon = moCommon
  moChildren.Add newChild
End Function

'cCommon
```

```
Public sName As String

'cChild
Private moCommon As cCommon
Public Name as String

Public Property Set oCommon(RHS as cCommon)
  Set moCommon = RHS
End Property

Public Property Get Path() as String
  Path = moCommon.Name & "/" & Name
End Property
```

As it stands that really only works for one level of parent child relations, but often we have an indefinite number of levels, for instance in a disk directory structure.

We can generalise this by recognizing that parents and children can actually be the same class and that the child doesn't care how the parent path is determined so long as it comes from the common object.

```
'cFolder
Private moChildren as Collection
Private moCommon as cCommon

Private Sub Class_Initialize()
  Set moChildren = New Collection
  Set moCommon = New cCommon
End Sub

Public Function NewFolder as cFolder
  Set NewFolder = New cFolder
  Set NewFolder.oCommon = moCommon
  moChildren.Add newFolder
End Function

Public Property Set oCommon(RHS as cCommon)
  Set moCommon.oCommon = RHS
End Property

Public Property Get Path() as String
  Path = moCommon.Path
End Property

Public Property Get Name() as String
  Name= moCommon.Name
End Property

Public Property Let Name(RHS As String)
  moCommon.Name = RHS
End Property

'cCommon
Private moCommon As cCommon
Public Name As String

Public Property Get Path() as String
  Path = "/" & Name
  if not moCommon is Nothing then
    ' has parent
    Path = moCommon.Path & Path
  End If
End Property
```

Now we can ask any object at any level of the structure for its full path and it will return it without needing a reference to its parent.

25.9.3 Exercises

- Create a simple program using the cfolder and cCommon classes and show that it works; that is, that it neither leaks memory nor gives the wrong answers for the *Path* property.

25.10 Errors and Exceptions

Before discussing the various kinds of errors we'll show how errors are handled in Visual Basic.

Visual Basic does not have exception classes, instead it has the older system of error codes. While this does make some kinds of programming awkward it really doesn't cause all that much trouble in well written programs. If your program doesn't rely on handling exceptions during normal operations you won't have much use for exception classes anyway.

However if you are part of a team that is creating a large program that consists of a large number of components (COM DLLs) it can be difficult to keep lists of error codes synchronized. One solution is to maintain a master list that is used by everyone in the project, another is to use exception classes after all. See VBCorLib[8] for an implementation in pure VB6 of many of the classes in the mscorlib.dll that provides the basis for programs created for Microsoft's .NET architecture.

There are two statements in Visual Basic that implement the error handling system:

- On Error Goto
- Err.Raise

The usual way of dealing with errors is to place an *On Error Goto* statement at the top of a procedure as follows:

```
On Error Goto EH
```

EH is a label at the end of the procedure. Following the label you place code that deals with the error. Here is a typical *error* handler*:*

```
    Exit Sub
  EH:
    If Err.Number = 5 Then
      FixTheProblem
      Resume
    End If
    Err.Raise Err.Number
  End Sub
```

There are several important things to notice about this:

8 http://www.kellyethridge.com/vbcorlib/index.shtml

- There is an *Exit Sub* statement immediately preceding the *error handler* label to ensure that when no error occurs the program does not fall into the *error handler*.
- The error code is compared to a constant and action taken depending on the result,
- The last statement *re-raises* the error in case there was no explicit handler,
- A resume statement is included to continue execution at the failed statement.

It might be that this is a perfectly useable handler for some procedure but it has some weak points:

- Use of a literal constant,
- The catch all *Err.Raise* statement doesn't provide any useful information, it just passes on the bare error code,
- *Resume* re-executes the failed statement.

There is no reason to ever use literal constants in any program. Always declare them either as individual constants or as enumerations. Error codes are best declared as enumerations like this:

```
Public Enum ErrorCodes
  dummy = vbObjectError + 1
  MyErrorCode
End Enum
```

When you want a new error code you just add it to the list. The codes will be assigned in increasing order. In fact you really never need to care what the actual number is.

You can declare the built in error codes in the just the same way except that you must explicitly set the values:

```
Public Enum vbErrorCodes
  InvalidProcedureCall = 5
End Enum
```

25.10.1 Taxonomy of Errors

There are broadly three kinds of errors:

Expected

Expected errors occur when the user or other external entity provides data that is clearly invalid. In such cases the user, whether a human or another program, must be informed about the error in the data not about where in the program it was discovered.

Failure to abide by the contract

a caller fails to supply valid arguments to a subroutine or a subroutine fails to provide a valid return value.

Unexpected

an error occurs that was not predicted by the specification of the program.

Expected errors are not really errors in the program but errors or inconsistencies in the data presented to the program. The program can must request that the user fix the data and retry the operation. In such cases the user has no use for a stack trace or other internal

arcana but has a considerable interest in clear and full description of the problem in external terms. The error report should point the user directly at the problem in the data and offer suggestions for fixing it. Don't just say "Invalid Data", say which piece of data is invalid, why it is invalid and what range of values can be accepted.

Contract failures usually indicate logic errors in the code. On the assumption that all the invalid data has been weeded out by the front end the program must work unless the program itself is faulty. See the #Assertions and Design By Contract[9] for more information on this topic.

Unexpected errors are the errors that most programmers concentrate on. In fact they are not terribly common, they only appear so because the contract between the various parts of the program is rarely spelled out. *Unexpected* errors differ from *Expected* errors in that the user cannot usually be expected to assist in immediate recovery from unexpected errors. The program then needs to present the user with all the gory details of the internal state of the program at the time that the error was discovered in a form that can easily be transmitted back to the maintainers of the program. A log file is ideal for this purpose. Don't just present the user with a standard message box because there is no practical way to capture the description so that it can be emailed to the maintainers.

25.10.2 Error Raising and Handling

Expected Errors

These are errors in the input data. From the point of view of the program they are not exceptions. The program should explicitly check for valid input data and explicitly inform the user when the data is invalid. Such checks belong in user interfaces or other parts of the program that are able to interact directly with the user. If the user interface is unable to perform the checks itself then the lower level components must provide methods that validate the data so that the user interface can make use of them. The method used to notify the user should depend on the severity and immediacy of the error as well as the general method of working with the program. For instance a source code editor can flag syntax errors without disturbing the creative flow of the user by highlighting the offending statements, the user can fix them at leisure. In other cases a modal message box might be the correct means of notification.

Contract Errors

These failures are detected by asserting the truth of the preconditions and postconditions of a procedure either by using the assertion statement or by raising an error if the conditions are not met. Such errors normally indicate faulty program logic and when they occur the report must sufficient information to enable the problem to be replicated. The report can easily be quite explicit about the immediate cause of the failure. When using Visual Basic it is important to remember that Debug.Assert statements are only executed in the IDE so it can be worthwhile raising erors instead in all but the most time critical routines.

9 Chapter 25.5 on page 126

Here is a simple, too simple, example of contract assertions:

```
Public Sub Reciprocal(n as Double) as Double
  Debug.Assert 0 <> n
  Reciprocal = 1 / n
End Sub
```

Visual basic will halt on the assertion of $n = 0$ so if you run your tests in the IDE you will be taken directly to the spot where the error is discovered.

Unexpected Errors

These are errors that are not caught by either validation of input data nor assertion of the truth of preconditions or postconditions. Like contract errors they indicate faulty logic in the program; of course the logic error can be in the validation of the input data or even in one of the preconditions or postconditions.

These errors need the most thorough report because they are obviously unusual and difficult or they would have been forseen in the validation and contract checks. Like contract failures the report should be logged to a text file so that it can easily be sent to the maintainers. Unlike the contract failures it won't be easy to tell what information is relevant so err on the safe side by including descriptions of all the arguments to the subroutines in the *Err.Raise* statements.

26 Idioms

An idiom is a sort of template or generic method of expressing an idea. In the same way that idioms in a human language make life easier for the both speaker and listener good idioms in computer programming make life easier for the programmer. Naturally what is idiomatic in one language might not be in another.

It is not necessary that there exist a physical template, an idiom can exist in your head and be so familiar that when faced with a problem that can be solved by application of an idiom you begin to type the code almost without noticing what you are doing.

Another word often used for ideas like this is *pattern*. However the word *pattern* has become so formalised and so strongly associated with object orientation that I feel the more generic word idiom is better also it is the word generally associated with the first major example that I will show.

An idiom can be language specific, library specific, domain specific, or any combination of the three. For instance if you are writing a program that deals with lists of things you might borrow some of the list processing idioms of Lisp even though Visual Basic Classic has no native list processing functions at all. By creating functions that manipulate lists rather than inlining the code to do it you will make your code more meaningful to someone reading it.

It is a commonplace that programmers feel ever so slightly godlike because they are in total control of their programs. I think that programmers would be better off emulating Shakespeare instead: when the English language of his day wasn't up to saying what he wanted he had no hesitation in extending both its stylistic range and its vocabulary.

26.1 Resource Acquisition Is Initialization - RAII

The name of this idiom makes it sound like a very complicated idea but in fact it is very simple and fits Visual Basic Classic very well. This idiom is a solution to problems such as setting and clearing busy flags, controlling access to files, setting the mouse icon and so on.

The basic idea is to use the constructor of an object to obtain or query a lock on some resource and to use the destructor to release that lock. In a language like Visual Basic which has deterministic finalization this produces very succinct code.

Let's take the mouse as an example.

26.1.1 Example: Mouse Busy Icon

It is very common in Windows programs to set the mouse icon to indicate that the program is busy and the the user must wait. A problem arises when you have lots of different pieces of code that can take a long time and lots of different code paths that invoke them. If the caller doesn't know that the process will be time consuming it won't know that the icon must be changed so we put code in the long running routine to change the mouse pointer. This works fine until we try to combine two or more such routines: which one gets to cancel the busy icon?

One common solution is to regard the mouse as a resource and control access to it with a pair of routines that maintain a count. One routine increments the count when the routine sets the busy icon and the other decrements and clears the icon when the count goes to zero. This works well until an exception occurs, then a routine might exit early and the routine that decrements the counter won't be called. Of course you could put another call in the error handler but that causes extra maintenance.

A better solution is to make use of the automatic termination events built in to Visual Basic. The basic idea is still to use a count but instead of guarding the count with a routine you guard it with an object.

Each procedure that wants to indicate that it is time consuming by showing the mouse busy icon should wrap its code in a with statement that creates a cMouseBusy instance by calling NewMouseBusy. When the routine finishes the cMouseBusy instance will be disposed of automatically, its Terminate event handler will be executed and the reference count will be reduced by one.

Because the program might have many forms open we must take account of the form when we set the busy icon because the MousePointer property belongs to a form not the application

So that we know which form's mouse pointer to change we must keep a separate count for each different client

Exercises

- Compile the example code and check that it works.
- Extend the code to allow for more than one type of busy icon. For instance an icon with a pointer that indicates that the user interface is still live and one without a pointer that indicates that the user must wait.
- Create a test program to demonstrate the new features.

Code

Just add this line as the first line of every function that should show the busy cursor:

```
Dim oMousebusy As cMouseBusy: Set oMousebusy = NewMouseBusy(Client)
```

or enclose the busy section in:

```
with NewMouseBusy(Client)
  ...
End With
```

Client is anything that has a mouseicon property

cMouseBusy
Objects of this class acquire the mouse busy status when created by a constructor called *NewMouseBusy*. They release it when they are terminated.

The busy status is simply a counter that is incremented each time a new cMouseBusy object is created and decremented when such an object is destroyed. If the counter increments to one then the mouse busy icon is set on the client object, when the counter decrements to zero the normal icon is set.

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
  Persistable = 0  'NotPersistable
  DataBindingBehavior = 0  'vbNone
  DataSourceBehavior  = 0  'vbNone
  MTSTransactionMode  = 0  'NotAnMTSObject
END
Attribute VB_Name = "cMouseBusy"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False

Option Explicit

Private moClient As Object

Friend Sub Initialize(ByRef roClient As Object)
  Set moClient = roClient
  If IncrefCount(roClient) = 1 Then
    roClient.MousePointer = vbHourglass
  End If
End Sub

Private Sub Class_Terminate()
  If DecRefCount(moClient) = 0 Then
    moClient.MousePointer = vbDefault
  End If
End Sub
```

modMouseBusy.bas
This module provide the constructor for cMouseBusy. This gives us the ability to pass arguments to the object when it is created.

```
Attribute VB_Name = "modMouseBusy"

Option Explicit

Private moReferenceCount As Dictionary

Public Function NewMouseBusy(ByRef roClient As Object) As cMouseBusy
  If moReferenceCount is Nothing then
    Set moReferenceCount = New Dictionary
```

```
  End If
  Set NewMouseBusy = New cMouseBusy
  NewMouseBusy.Initialize roClient
```

Remember that Visual Basic is single threaded and that the User Interface will not update until all code stops executing unless you call DoEvents so do this to make sure that the change to the mouse icon is visible.

```
  DoEvents
End Function
```

This function increments the count for the given object and returns the new count.

```
Public Function IncrefCount(ByRef roClient As Object) As Long
  Dim lRefCount As Long
  IncrefCount = 1 + moReferenceCount(roClient)
  moReferenceCount(roClient) = IncrefCount
End Function
```

This function decrements the count and returns the new count. Note that if the count goes to zero then the entry is removed from the list; this must be done because the *keys* are objects and they will not terminate if we do not release the reference.

```
Public Function DecRefCount(ByRef roClient As Object) As Long
  DecRefCount = moReferenceCount(roClient) - 1
  If DecRefCount = 0 Then
    moReferenceCount.Remove roClient
  Else
    moReferenceCount(roClient) = DecRefCount
  End If
End Function
```

frmTestMouseBusy

This form lets you test the mouse busy class. Just click the buttons and watch the mouse cursor.

```
VERSION 5.00
Begin VB.Form frmTestMouseBusy
   Caption         =   "frmTestMouseBusy"
   ClientHeight    =   2175
   ClientLeft      =   60
   ClientTop       =   360
   ClientWidth     =   4140
   LinkTopic       =   "Form1"
   ScaleHeight     =   2175
   ScaleWidth      =   4140
   StartUpPosition =   3  'Windows Default
   Begin VB.CommandButton Command2
      Caption         =   "Command2"
      Height          =   735
      Left            =   2160
      TabIndex        =   1
      Top             =   480
      Width           =   1215
   End
   Begin VB.CommandButton Command1
      Caption         =   "Command1"
      Height          =   735
      Left            =   480
      TabIndex        =   0
```

```
      Top             =   480
      Width           =   1215
   End
End
Attribute VB_Name = "frmtestMouseBusy"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit


Private Sub xb1(ByRef delay As Long)
  With NewMouseBusy(Me)
    Dim t As Double
    t = Timer
    Do While t + delay > Timer
      DoEvents
    Loop
    xBug
  End With

End Sub


Private Sub xBug()
  With NewMouseBusy(Me)
    Dim t As Double
    t = Timer
    Do While t + 3 > Timer
      DoEvents
    Loop
    Debug.Print 1 / 0
  End With

End Sub

Private Sub Command1_Click()
  On Error Resume Next
  xb1 3
End Sub

Private Sub Command2_Click()
  On Error Resume Next
  xb1 5
End Sub
```

prjTestMouseBusy.vbp

The project file for the test. Note that you need a reference to the Microsoft Scripting Runtime library to provide the Dictionary class.

```
Type=Exe
Reference=*\G{00020430-000
0-0000-C000-000000000046}#2.0#0#..\..\..\..\..\..\WINNT\System32\stdole2.tlb#OLE
 Automation
Reference=*\G{420B2830-E718-11C
F-893D-00A0C9054228}#1.0#0#..\..\..\..\..\..\WINNT\System32\scrrun.dll#Microsoft
 Scripting Runtime
Class=cMouseBusy; cMouseBusy.cls
Module=modMouseBusy; modMouseBusy.bas
Form=frmtestMouseBusy.frm
Startup="frmTestMouseBusy"
HelpFile=""
Command32=""
Name="prjTestMouseBusy"
```

```
HelpContextID="0"
CompatibleMode="0"
MajorVer=1
MinorVer=0
RevisionVer=0
AutoIncrementVer=0
ServerSupportFiles=0
VersionCompanyName="ABB"
CompilationType=0
OptimizationType=0
FavorPentiumPro(tm)=0
CodeViewDebugInfo=0
NoAliasing=0
BoundsCheck=0
OverflowCheck=0
FlPointCheck=0
FDIVCheck=0
UnroundedFP=0
StartMode=0
Unattended=0
Retained=0
ThreadPerObject=0
MaxNumberOfThreads=1
```

26.2 References

- The RAII (Resource Acquisition Is Initialisation) Programming Idiom[1].

1 http://www.hackcraft.net/raii/

27 Optimizing Visual Basic

Optimization is the art and science of making your program faster, smaller, simpler, less resource hungry, etc. Of course faster often conflicts with simpler and smaller so optimization is a balancing act.

This chapter is intended to contain a toolkit of techniques that you can use to speed up certain kinds of program. Each technique will be accompanied by code that demonstrates the improvement. Please note that the code is deliberately sub-optimal for two reasons: optimized code is almost always harder to understand and the purpose is that the reader should exercise the techniques in order to obtain greater understanding. You should always be able to rewrite the code presented to obtain greater improvements but this will sometimes be accompanied by a reduction in readability. In code that is speed critical choose speed over clarity, in code that is not speed critical you should always choose the most maintainable style; remember that another programmer coming to your code years later relies on you to make the code say what it means. When you optimize the code you will often need to be extra careful in adding comments so as to ensure that someone maintaining the code doesn't throw away your hard earned speed improvement because he doesn't understand why you use what seems like an overly complicated method.

If you change code to gain speed you should write tests that demonstrate that the new code really is faster than the old. Often this can be as simple as recording the time before and after a procedure and taking the difference. Do the test before you change the code and after and see if the improvement was worthwhile.

Remember that the code has to work first. Make it work, then make it work faster.

Code that slowly gets the right answer is almost invariably preferable to code that quickly gets the wrong answer. This is especially true if the answer is a small part of a larger answer so that errors are hard to spot.

Ensure that things work by writing tests, validation functions and assertions. When you change the code to make it faster the results of the tests, validation functions and assertions should not have changed.

27.1 Integers and Longs

In VB6 it is generally the case that a program in which all whole number variables are declared as Long will run faster than than the same program using Integer.

Use:

```
Dim I as Long
```

not:

```
Dim I as Integer
```

However, it is important to realize that the final arbiter of speed is measurement of performance under real life conditions; see the tests section below.

The only reason to use Integer is when passing arguments by reference to functions defined in components that you cannot alter. For instance most of the events raised by the VB6 GUI use Integer arguments instead of Long. This is also the case with some third party DLLs. Most Windows API functions require *Long* arguments.

27.1.1 Tests

Here is a simple test that illustrates that things are not always as they seem in the world of optimization. The test is a simple module that executes two subroutines. The two routines are identical except that in one all the whole number variables are declared as Long and in the other they are all declared as Integer. The routines simply increment a number repeatedly. When executed inside the VB6 IDE on my computer the results are (three consecutive runs in seconds):

```
Long         11.607
Integer       7.220
Long         11.126
Integer       7.211
Long         11.006
Integer       7.221
```

which appears to contradict my assertion that Longs are faster than Integers. However when compiled and executed outside the IDE the results are:

```
Long         0.711
Integer      0.721
Long         0.721
Integer      0.711
Long         0.721
Integer      0.721
```

Notice that when running compiled the times for Long and Integer are indistinguishable. This illustrates several important points about optimization and benchmarking.

Timing jitter

timing an algorithm on a pre-emptive multitasking operating system is always an exercise in statistics because you cannot (not in VB6 at least) force the OS to not interrupt your code which means that timing using the simple technique of these tests includes time spent in other programs or at least in the System Idle loop.

Know what you are measuring

the times measured in the IDE are at least ten times longer than those measured when the code is compiled to native code (optimize for *fast* code, integer bounds checking *on* in this case). The relative difference in timing for Long and Integer differs between compiled and interpreted code.

For more benchmarks see Donald Lessau's excellent site: VBSpeed[1].

Here is the code:

```
Option Explicit

Private Const ml_INNER_LOOPS As Long = 32000
Private Const ml_OUTER_LOOPS As Long = 10000

Private Const mi_INNER_LOOPS As Integer = 32000
Private Const mi_OUTER_LOOPS As Integer = 10000


Public Sub Main()

  Dim nTimeLong As Double
  Dim nTimeInteger As Double

  xTestLong nTimeLong
  xTestInteger nTimeInteger

  Debug.Print "Long", Format$(nTimeLong, "0.000")
  Debug.Print "Integer", Format$(nTimeInteger, "0.000")

  MsgBox "   Long: " & Format$(nTimeLong, "0.000") & vbCrLf _
         & "Integer: " & Format$(nTimeInteger, "0.000")

End Sub


Private Sub xTestInteger(ByRef rnTime As Double)

  Dim nStart As Double

  Dim iInner As Integer
  Dim iOuter As Integer
  Dim iNum As Integer

  nStart = Timer
  For iOuter = 1 To mi_OUTER_LOOPS
    iNum = 0
    For iInner = 1 To mi_INNER_LOOPS
      iNum = iNum + 1
    Next iInner
  Next iOuter

  rnTime = Timer - nStart

End Sub


Private Sub xTestLong(ByRef rnTime As Double)

  Dim nStart As Double
```

1 http://xbeat.net/vbspeed/index.htm

```
  Dim lInner As Long
  Dim lOuter As Long
  Dim lNum As Long

  nStart = Timer
  For lOuter = 1 To ml_OUTER_LOOPS
    lNum = 0
    For lInner = 1 To ml_INNER_LOOPS
      lNum = lNum + 1
    Next lInner
  Next lOuter

  rnTime = Timer - nStart

End Sub
```

27.2 Strings

If you have a lot of code that does a lot of string concatenation you should consider using a string builder. A string builder is usually provided as a class but the principle is quite simple and doesn't need any object oriented wrapping.

The problem that a string builder solves is the time used in repeatedly allocating and deallocating memory. The problem arises because VB strings are implemented as pointers to memory locations and every time you concatenate two strings what actually happens is that you allocate a new block of memory for the resulting string. This happens even if the new string replaces the old as in the following statement:

```
s = s & "Test"
```

The act of allocating and deallocating memory is quite expensive in terms of CPU cycles. A string builder works by maintaining a buffer that is longer than the actual string so that the text that is to be added to the end can simply be copied to the memory location. Of course the buffer must be long enough to accommodate the resulting string so we calculate the length of the result first and check to see if the buffer is long enough; if it is not we allocate a new longer string.

The code for a string builder can be very simple:

```
Private Sub xAddToStringBuf(ByRef rsBuf As String, _
                            ByRef rlLength As Long, _
                            ByRef rsAdditional As String)

  If (rlLength + Len(rsAdditional)) > Len(rsBuf) Then
    rsBuf = rsBuf & Space$(rlLength + Len(rsAdditional))
  End If
  Mid$(rsBuf, rlLength + 1, Len(rsAdditional)) = rsAdditional
  rlLength = rlLength + Len(rsAdditional)

End Sub
```

This subroutine takes the place of the string concatenation operator (&). Notice that there is an extra argument, *rlLength*. This is necessary because the length of the buffer is not the same as the length of the string.

We call it like this:

```
dim lLen as long
xAddToString s, lLen, "Test"
```

lLen is the length of the string. If you look at the table of execution times below you can see that for values of Count up to about 100 the simple method is faster but hat above that the time for simple concatenation increases roughly exponentially whereas that for the string builder increases roughly linearly (tested in the IDE, 1.8GHz CPU, 1GB ram). The actual times will be different on your machine but the general trend should be the same.

It is important to look critically at measurements like this and try to see how, or if, they apply to the application that you are writing. If you build long text strings in memory the string builder is a useful tool but if you only concatenate a few strings the native operator will be faster and simpler.

27.2.1 Tests

Concatenating the word *Test* repeatedly gives these results

Times in seconds

Count	Simple	Builder
10	0.000005	0.000009
100	0.000037	0.000043
1000	0.001840	0.000351
5000	0.045	0.002
10000	0.179	0.004
20000	0.708	0.008
30000	1.583	0.011
40000	2.862	0.016
50000	4.395	0.019
60000	6.321	0.023
70000	13.641	0.033
80000	27.687	0.035

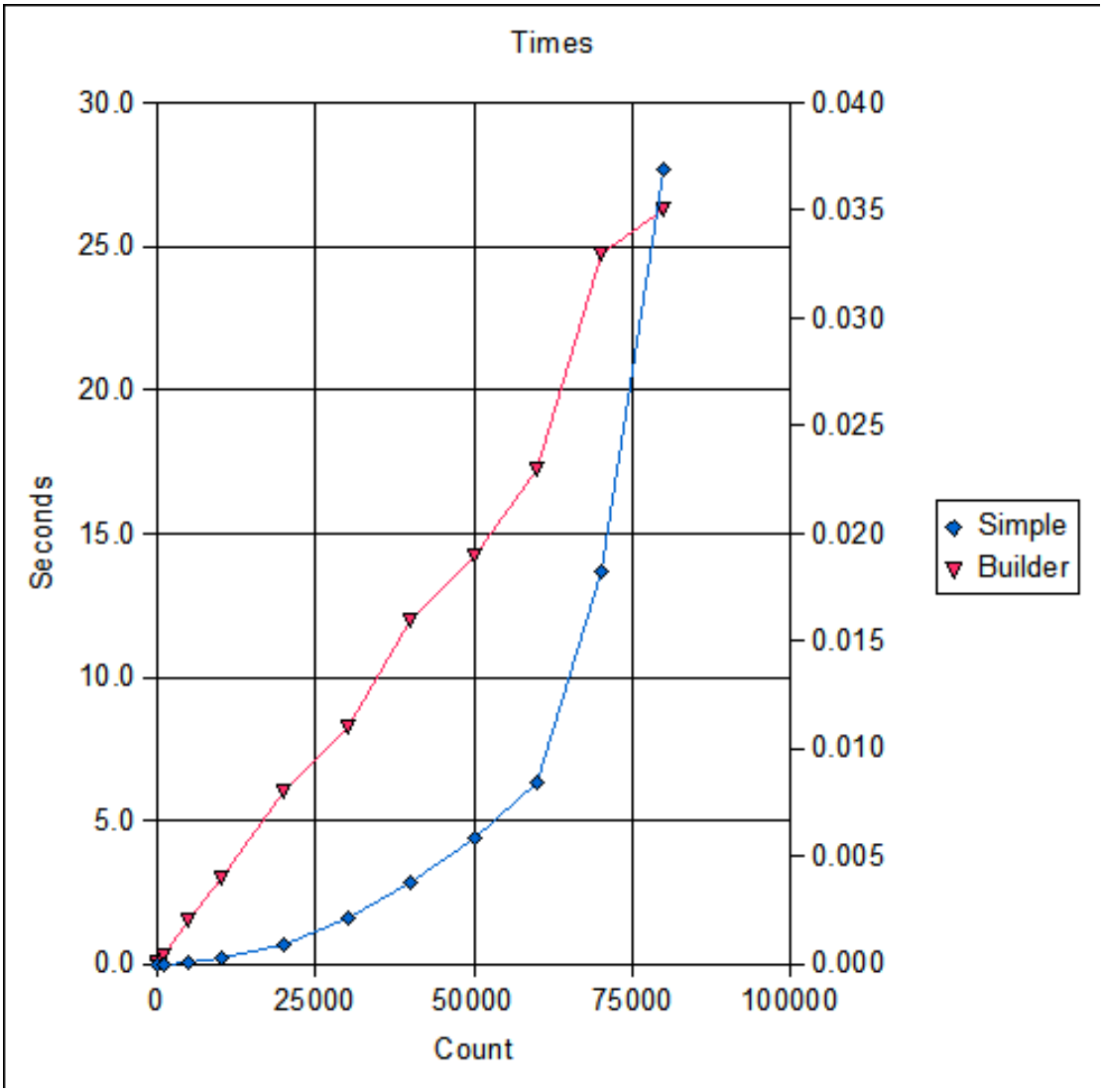


Figure 8

27.3 ByRef versus ByVal

Many books on VB tell programmers to always use ByVal. They justify this by saying that it is both faster and safer.

You can see that ByVal is slightly faster by the results of the Tests[2]:

Empty functions:

```
xTestByRef     13.4190000000017
xTestByVal     13.137999999999
```

With a simple arithmetic expression in the function:

```
xTestAddByRef  15.7870000000039
xTestAddByVal  15.3669999999984
```

You can also see for that the difference is slight. Be careful when interpreting coding benchmarks, even the simplest can be misleading and it is always wise to profile the real code if you think you have a bottle-neck.

The other claim is that ByVal is safer. This is usually explained by saying that a function in which all arguments are declared ByVal cannot alter the values of the variables in the caller because VB makes copies instead of using pointers. The programmer is then free to make use of the incoming arguments in any way he or she sees fit, in particular they can be assigned new values without disturbing the original values in the caller.

The safety provided by this is generally outweighed by the fact that the compiler is then unable to enforce type checking. In Visual Basic Classic variables are automatically converted between *string* and *numeric* types when necessary and possible. If arguments are declared ByVal then the caller can provide a *String* where a *Long* was expected. The compiler will silently compile the necessary instructions to convert the *String* to a *Long* and carry on. At run time you might, or might not, discover the problem. If the string contained a number then it will be converted and the code will 'work', if it doesn't then the code will fail at the point where the function is called.

```
Function IntByVal(ByVal z as Double) as Long
  IntByVal = Int(z)
End Function

Function IntByRef(ByRef z as Double) as Long
  IntByRef = Int(z)
End Function

Dim s As String
's = "0-471-30460-3"
s = "0"
Debug.Print IntByVal(s)
```

2 Chapter 27.3.1 on page 152

```
Debug.Print IntByRef(s)
```

If you try to compile the little snippet of code above you will see that the compiler stops on the line:

```
Debug.Print IntByRef(s)
```

it hightlights the *s* and shows a message box saying 'ByRef argument type mismatch'. If you now comment out that line and run again you will not get an error. Now uncomment this line:

```
s = "0-471-30460-3"
```

and comment out this one:

```
's = "0"
```

Run the program again. Now the program fails but only at run time with 'Runtime error 13: Type mismatch'.

The moral is:

- use ByRef to cause the compiler to typecheck the arguments in function calls,
- never assign to the arguments of a function unless they are *output* parameters.

Look at the Procedure Parameters[3] section in the Coding Standards[4] chapter for some suggestions about how to name the arguments so that it is always clear which are *in* and which are *out* parameters.

27.3.1 Tests

With Empty Functions

```
Option Explicit

Private Const mlLOOPS As Long = 100000000
Private mnStart As Double
Private mnFinish As Double

Public Sub main()
  xTestByRef 1#, 2#
  Debug.Print "xTestByRef", mnFinish - mnStart
  xTestByVal 1#, 2#
  Debug.Print "xTestByVal", mnFinish - mnStart
End Sub
```

3 Chapter 31.36 on page 212
4 Chapter 30.5 on page 186

```
Private Sub xTestByRef(ByRef a As Double, ByRef b As Double)
  Dim lLoop As Long
  Dim n As Double
  mnStart = Timer
  For lLoop = 1 To mlLOOPS
    n = xByRef(a, b)
  Next lLoop
  mnFinish = Timer
End Sub

Private Sub xTestByVal(ByVal a As Double, ByVal b As Double)
  Dim lLoop As Long
  Dim n As Double
  mnStart = Timer
  For lLoop = 1 To mlLOOPS
    n = xByVal(a, b)
  Next lLoop
  mnFinish = Timer
End Sub

Private Function xByRef(ByRef a As Double, ByRef b As Double) As Double
End Function

Private Function xByVal(ByVal a As Double, ByVal b As Double) As Double
End Function
```

With Simple Arithmetic

```
Attribute VB_Name = "modMain"
Option Explicit

Private Const mlLOOPS As Long = 100000000
Private mnStart As Double
Private mnFinish As Double

Public Sub main()
  xTestAddByRef 1#, 2#
  Debug.Print "xTestAddByRef", mnFinish - mnStart
  xTestAddByVal 1#, 2#
  Debug.Print "xTestAddByVal", mnFinish - mnStart
End Sub

Private Sub xTestAddByRef(ByRef a As Double, ByRef b As Double)
  Dim lLoop As Long
  Dim n As Double
  mnStart = Timer
  For lLoop = 1 To mlLOOPS
    n = xAddByRef(a, b)
  Next lLoop
  mnFinish = Timer
End Sub

Private Sub xTestAddByVal(ByVal a As Double, ByVal b As Double)
  Dim lLoop As Long
  Dim n As Double
  mnStart = Timer
  For lLoop = 1 To mlLOOPS
    n = xAddByVal(a, b)
  Next lLoop
  mnFinish = Timer
End Sub

Private Function xAddByRef(ByRef a As Double, ByRef b As Double) As Double
  xAddByRef = a + b
```

```
End Function

Private Function xAddByVal(ByVal a As Double, ByVal b As Double) As Double
  xAddByVal = a + b
End Function
```

27.4 Collections

Collections are very useful objects. They allow you to write simpler code than would otherwise be the case. For instance if you need to hold on to a list of numbers given to you by the user you probably won't know in advance how many will be supplied. This makes it difficult to use an array because you must either allocate an unnecessarily large array so that you are sure that no reasonable user would supply more or you must continually Redim[5] the array as new numbers come in.

A Collection lets you avoid all that because it expands as necessary. However, this convenience comes at the cost of increased runtime under some circumstances. For many, perhaps most, programs this price is perfectly acceptable but some programs take so long to run that you must try to squeeze the last drop of performance out of every line.

This frequently occurs in programs doing scientific computations (don't tell me C would be a better language for such things because the same constraints and optimizations apply to C and all other languages as well).

One of the reasons a Collection is a convenient tool is because you can use a string as a key and retrieve items by key instead of by index. But every time you ask for an item in a collection the collection must first figure out whether you have supplied an Integer (or Byte or Long) or a String which of course takes a small but finite amount of time. If your application makes no use of the ability to look up items by String key you will get faster programs by using arrays instead because VB doesn't need to waste time checking to see if you supplied a String key instead of a whole number index.

If you want a Collection to simply hold a list of items without keys then you can emulate this behaviour with an array as follows (note that this code is not optimal, optimizing it is left as an exercise for the student):

```
Public Sub Initialize(ByRef a() As Variant, _
                      ByRef Count As Long, _
                      ByRef Allocated As Long)
  Allocated = 10
  Redim a(1 to Allocated)
  Count = 0
End Sub

Public Sub Add(ByRef NewItem as Variant, _
               ByRef a() As Variant, _
               ByRef Count As Long, _
               ByRef Allocated As Long)
  Count = Count + 1
  If Allocated < Count then
```

5 http://en.wikibooks.org/wiki/Programming%3AVisual%20Basic%20Class%2FKeywords%23Redim

```
      Allocated = Count
      Redim Preserve a(1 to Allocated)
    End If
    a(Count) = NewValue
  End Sub
```

To use the above code you must declare an array of Variants and two Longs. Call the Initialize sub to start it all off. Now you can call the Add sub as often as you like and the array will be extended as necessary.

There are a number of things to note about this seemingly simply subroutine:

- Once the array size exceeds 10 a new array is allocated each time an item is added (replacing the original),
- The size of the array (recorded in the Allocated variable) is always the same as the Count when Count exceeds 10,
- No provision is made to delete items,
- the items are stored in the same order as they are added.
- all arguments are declared Byref

It is unlikely that any programmer would want to include exactly this routine in production code. Some reasons why are:

- NewItem is declared as Variant but the programmer usually knows the type,
- The initial allocation is hard coded using a literal integer,
- The name of the subroutine is too simple, it will probably clash with others in the same namespace.
- No provision is made for removing items.
- Three separate pieces of information must be kept together but they are not bound together.

The performance of the emulated Collection depends on the relative frequency of calls to Add, Item and Remove (see #Exercises[6]). An optimized version must be optimized for the use to which it will be put. If no lookups by key are performed there is no need to provide a function for it and there is especially no need to provide data structures to make it efficient.

27.4.1 Exercises

- Extend the code presented in this section to emulate the Collection class in more detail. Implement Item and Remove methods, see the VB help files for details of the exact method declarations but do not feel obliged to replicate them exactly.
- Write a test program to exercise all the features and time them so that you can tell when to use a Collection and when to use your emulated version.
- You should be able to think of at least two distinct implementations of Remove. Think about the consequences of different implementations.

6 Chapter 27.7.1 on page 163

- Write an explicit description of the requirements that your version of the Collection class satisfies.
- Compare the behaviour of the built in Collection class and the new class, note any differences and give examples of uses where the differences do and do not matter.
- If you haven't already done so implement lookup by key.
- Explain why emulating all of the features of the Collection class using exactly the same interface is unlikely to yield significant improvements in performance for at least one specific use case.

27.5 Dictionaries

In VB a dictionary is actually supplied by the Scripting Runtime component. It is rather like a Collection but also provides a means to retrieve the keys that were used. Also, unlike the Collection, the keys can be any datatype including objects. The principal differences between Collections and Dictionaries are as follows:

- Dictionaries have Keys and Items properties which return variant arrays
- Keys can be any datatype not just strings
- Enumeration using For Each returns the Keys not the Items
- There is a built in Exists method.

As with Collections the convenience of Dictonaries is sometimes outweighed by their run time expense. One reason Dictionaries are frequently used instead of Collections is simply because of the Exists method. This allows you to avoid overwriting existing data or to avoid attempts to access non-existent data. However if the number of data items is small then a simple linear search can be faster. A small number in this case might be as many as 50 items. You can write a simple class that can be used instead of a Dictionary as shown below; note that no attempt is made to replicate all the behaviour of the dictionary and that this is done intentionally.

```
'cDict.cls
Option Explicit

Private maItems() As String
Private mlAllocated As Long
Private mlCount As Long

Public Sub Add(Key As String, Item As String)

  mlCount = mlCount + 1
  If mlAllocated < mlCount Then
    mlAllocated = mlAllocated + mlCount
    ReDim Preserve maItems(1 To 2, 1 To mlAllocated)
  End If
  maItems(1, mlCount) = Key
  maItems(2, mlCount) = Item

End Sub

Public Property Get Count() As Long
  Count = mlCount
```

```
End Property

Public Function Exists(Key As String) As Boolean

  Exists = IndexOf(Key)

End Function

Public Function IndexOf(Key As String) As Long

  For IndexOf = 1 To mlCount
    If maItems(1, IndexOf) = Key Then
      Exit Function
    End If
  Next IndexOf
  IndexOf = 0

End Function

Public Property Let Item(Key As String, RHS As String)

  Dim lX As Long
  lX = IndexOf(Key)
  If lX Then
    maItems(lX, 2) = RHS
  Else
    Add Key, RHS
  End If

End Property

Public Property Get Item(Key As String) As String
  Item = maItems(IndexOf(Key), 2)
End Property

Public Sub Remove(Key As String)

  Dim lX As Long
  lX = IndexOf(Key)
  maItems(1, lX) = maItems(1, mlCount)
  maItems(2, lX) = maItems(2, mlCount)
  mlCount = mlCount - 1

End Sub

Public Sub RemoveAll()
  mlCount = 0
End Sub

Public Sub Class_Initialize()
  mlAllocated = 10
  ReDim maItems(1 To 2, 1 To mlAllocated)
End Sub
```

A simple test routine can be used to demonstrate that this class is faster than the VB dictionary for certain tasks. For instance adding 32 items to the dictionary and then removing them again is faster using cDict but if you double the number of items the VB Dictionary is better. The moral is: choose the correct algorithm for the task at hand.

Here is the test routine:

```
Option Explicit

Public gsModuleName As String


Private mnStart As Double
Private mnFinish As Double

Private Const mlCount As Long = 10000


Public Sub main()

  Dim litems As Long
  litems = 1
  Do While litems < 100
    litems = litems * 2
    Debug.Print "items=", litems
    Dim lX As Long
    mnStart = Timer
    For lX = 1 To mlCount
      xTestDictionary litems
    Next lX
    mnFinish = Timer
    Debug.Print "xTestDictionary", "Time: "; Format$(mnFinish - mnStart,
"0.000")

    mnStart = Timer
    For lX = 1 To mlCount
      xTestcDict litems
    Next lX
    mnFinish = Timer
    Debug.Print "xTestcDict     ", "Time: "; Format$(mnFinish - mnStart,
"0.000")
  Loop


End Sub



Private Sub xTestDictionary(ByRef rlItems As Long)

  Dim d As Dictionary
  Set d = New Dictionary

  Dim lX As Long
  Dim c As Double
  For lX = 1 To rlItems
    d.Add Str$(lX), Str$(lX)
  Next lX
  For lX = 1 To rlItems
    d.Remove Str$(lX)
  Next lX

End Sub


Private Sub xTestcDict(ByRef rlItems As Long)

  Dim d As cDict
  Set d = New cDict

  Dim lX As Long
  Dim c As Double
  For lX = 1 To rlItems
```

```
    d.Add Str$(lX), Str$(lX)
  Next lX
  For lX = 1 To rlItems
    d.Remove Str$(lX)
  Next lX

End Sub
```

And here are the results from my PC in the IDE (seconds):

```
items=         2
xTestDictionary             Time: 1.602
xTestcDict                  Time: 0.120
items=         4
xTestDictionary             Time: 1.663
xTestcDict                  Time: 0.200
items=         8
xTestDictionary             Time: 1.792
xTestcDict                  Time: 0.361
items=         16
xTestDictionary             Time: 2.023
xTestcDict                  Time: 0.741
items=         32
xTestDictionary             Time: 2.504
xTestcDict                  Time: 1.632
items=         64
xTestDictionary             Time: 3.455
xTestcDict                  Time: 4.046
items=         128
xTestDictionary             Time: 5.387
xTestcDict                  Time: 11.437
```

27.5.1 Exercises

- Write a version of the cDict class that allows storage of other data types, using Variant arguments for instance,
- Check the performance using a similar test routine.
- Write a new test routine that performs other tests. Does the relative performance of the new classes and VB Dictionary change?
- What happens if you try to retrieve or remove an item that does not exist? How does the behaviour compare with that of the Dictionary?

27.6 The Debug Object

The Debug object has two methods:

Print

prints its arguments to the immediate window,

Assert

pauses the program if its argument is *false*.

Both of these only have an effect when running in the IDE; at least that is the conventional wisdom. Unfortunately it is not quite true of Debug.Print. This method doesn't print anything when the program is running as a compiled executable but if the arguments are function calls they are still evaluated. If the function call is very time consuming you will find that the compiled version doesn't run as fast as expected.

There two things that can be done:

- Remove all Debug.Print statements before shipping the product,
- Only use variables or constants as arguments to Debug.Print

If your program is both very CPU intensive and under continual development the second might be preferable so that you don't have to keep adding and removing lines.

Debug.Assert doesn't suffer from this problem so it is perfectly safe to assert the truth of complicated and time consuming functions. The assertion arguments will only be evaluated when running in the IDE.

27.6.1 Exercises

- Write a short program to demonstrate that Debug.Print evaluates functions even when compiled,
- Modify it to show that Debug.Assert does not suffer from this problem,
- Show that Debug.Print has zero execution time in the compiled version if the arguments are constants or variables.

27.7 Object Instantiation

Long words for simple concepts. This section deals with the runtime cost of creating objects. The jargon phrase for this is *object instantiation* which means creating an *instance* of a *class*. An *instance* bears the same relation to a *class* as a *machine* bears to *plans* for it. For any given class there can be as many instances as you like.

If an object takes a long time to construct and you create and destroy a lot of them during the running of the program you might save some time by not destroying them but putting them on a list of ready made objects for later use.

Imagine a program that simulates an animal ecology. There could be, say, two classes of animals: herbivores and carnivores.

If you want to see any results in your lifetime the simulation must obviously run faster than real life speed so a lot of herbivores in particular will be born, breed and be eaten. If each of these is represented by an object that is destroyed when the animal it represents is killed the system will be repeatedly allocating and deallocating memory which is a relatively expensive business in VB6. In such a program you know that new objects of the same kind are continually needed so you can avoid some of the memory allocation overhead by reusing dead objects instead of destoying them and creating new ones. There are many ways of doing this. Which one you choose depends on how many different classes of object you

have. If there are very few then you can create separate code for each, this can then be very highly tuned. On the other hand if you have dozens of different classes (perhaps you have extended the simulation to include competing herbivores) you will quickly find that you have a maintenance problem.

The solution is to create a class that describes a general purpose object pool and an Interface[7] that can be implemented by each of the classes.

Before I describe the class and the interface here is a summary of the requirements:

- Only one pool class needed,
- The classes being 'pooled' should need small modifications to only the initialization and terminaion code to fit the *pooling* concept,
- The code that uses the pooled objects should not have to change except for calling the pool's GetObject function instead of using the New operator.

The technique hinges on the fact that VB6 has *deterministic finalization*. Another piece of jargon that simply means that VB6 destroys (finalizes) objects as soon as they become unused. Every time VB6 decides that an object is no longer in use it calls the Class_Terminate method of that object. What we can do is add a simple piece of code to each class that puts the terminating object on a list of available objects. VB will see that the object is now in use and not deallocate the memory that it used. Later on, instead of using the *New* operator to create new object we ask the object pool to give us one that had fallen out of use. Even if the object takes very little time to set up this will be quicker than using *New* because it avoid both the allocation and deallocation of memory.

Here is an example object pool class and a class that can use it:

```
'ObjectPool.cls
Private moPool as Collection
Public oTemplate As Object

Private sub Class_Initialize()
  set moPool = New Collection
End Sub

Public Function GetObject() as Object
  if moPool.Count then
    Set GetObject = moPool(1)
    moPool.Remove(1)
  Else
    Set GetObject = oTemplate.NewObject
  End If
End Function

Public Sub ReturnToPool(Byref oObject as Object)
  moPool.Add oObject
End Sub
```

To use this class declare a public variable in a *bas* module and assign a new ObjectPool object to it:

7 http://en.wikibooks.org/wiki/..%2FInterfaces

```
'modPool.bas
Public goPool As ObjectPool

Public Sub Initialize()
  Set goPool = New ObjectPool
  Set goPool.oTemplate = New Herbivore
End Sub
```

Now modify your Herbivore class by adding a call to ReturnToPool to the Class_Terminate method and adding a NewObject method:

```
Private Sub Class_Terminate()
  goPool.ReturnToPool Me
End Sub

Public Function NewObject() as Object
  Set NewObject = New Herbivore
End Function
```

For some simple scenarios this might even work. However, there are several flaws, and at least one is a major problem. The problem is that the object you get doesn't necessarily look like a shiny new one. Now for some applications this doesn't matter because the client code will reinitialise everything anyway but many programs rely on the feature of the language that automatically sets newly allocated variables to zero. In order to satisfy the requirement for minimum changes in the client we should extend both ObjectPool and the objects that are pooled by adding a ReInitialize method to the Herbivore:

```
Public Sub ReInitialize()
  ' Do whatever you need to do to make the object
  ' look like new (reset some attributes to zero,
  ' empty strings, etc).
End Sub
```

Don't do any more work in *ReInitialize* than absolutely necessary. For instance if one of the attributes of the object is a dynamically allocated array it might not be necessary to clear it; it might be enough to set a flag or counter to indicate that no array elements are actually in use.

Now modify the GetObject method of ObjectPool:

```
Public Function GetObject() as Object
  if moPool.Count then
    Set GetObject = moPool(1)
    GetObject.ReInitialize
    moPool.Remove(1)
  Else
    Set GetObject = oTemplate.NewObject
  End If
End Function
```

Now everywhere that you use New Herbivore use goPool.GetObject instead. If the Herbivore object has references to other objects you might, or might not, want to release those references by setting them to Nothing in the Class_Terminate method. It depends on the

behaviour of the objects and the rest of your program, generally you should put off doing expensive operations for as long as possible.

The use of an *object pool* can improve the performance of certain types of program without requiring the programmer to radically change the program design. However don't forget that you might be able to get similar or greater improvements by using a better algorithm; again timing tests are the key to knowledge in this area. Don't assume that you know where the bottleneck is, demonstrate where it is by profiling the program.

27.7.1 Exercises

- Write a simple program that creates and destroys lots of objects and time it. Now modify it to use ObjectPool.
- Define an interface for pooled objects and implement it. Does eliminating the use of *As Object* change the performance?
- Notice that the moPool collection is used simply as a FIFO stack, that is no use is made of the ability to find items by key. Is there an alternative that would be faster?
- Is the FIFO behaviour of the stack important, that is, is it a deliberate feature or simply irrelevant?

Tip #1 – Move as much code as possible outside of loops. Loops are always the most important thing to optimize in your code. Always try to move as much as you can out of a loop. That way code doesn't get repeated and saves you some cpu cycles. A simple example:

Code: for i = 1 to 50 x = b ' Stays the same with every loop, get it outside of the loop! k = j + i next iChange that to:

Code: x = b 'Moved outside the loop

for i = 1 to 50 k = j + 1 next i That may seem like a no-brainier but you would be surprised about how many programmers do that. The simple rule is, if it doesn't change with every loop iteration then move it outside of the loop as it doesn't need to be there. You only want to include code inside a loop that MUST be there. The more instructions you can clear out of a loop the faster we can run it.

Tip #2 – Loop Unrolling Loop unrolling can eliminate some compare and jump instructions. (Compare and jump instructions are used to create loops, you don't see them in visual basic, its behind the scenes stuff that you learn in ASM.) It also takes advantage of the ability of modern cpus that can fetch several instructions at a time. In a nutshell you get a good speed boast by unrolling loops.

But there is something we need to know about loop unrolling. The largest bottleneck on modern computers is memory. So the designers of CPU's like Intel and AMD addressed this problem by using a cache on their cpus. This is basically a memory location that is accessed much faster by the CPU then standard memory. You want your unrolled loop to fit in that cache, if it doesn't then it could slow down your code. So you may want to experiment with gettickcount when you unroll you're loop.

Example Loop:

Code: For i = 1 To 100

```
        b = somefun(b)
```

Next i unrolled Example:

Code: For i = 1 To 100 Step 2

```
        b = somefun(b)
        b = somefun(b)
```

Next i You can get up to a 25% gain in speed depending on what you are doing, you just have to experiment with this.

Tip #3 – Avoid dividing if possible. A divide instruction is one of the most if not the most expensive instruction you can perform on a CPU. It is faster to multiply then divide!

Code: B = 40 / 2

is slower then

Code: b = 40 * 0.5 You can also develop some interesting algorithms using subtraction to get your result that is much faster then using division. If your using division in a loop, it is a must to change it to speed up your code. (I was going to also recommend trying shifting the bits for division but I forgot some versions of visual basic doesn't include the shift operator).

Tip #4 – In a nested conditional branch such as select case and nested if statements, put the things that are most likely to be true first in the nest, with the least likely things last.

Tip #5 – Avoid use of variant variables. The variant variable is all nice when you are new to visual basic, but its a habit you need to break. A variant variable is converted into its proper data type at runtime which can be very expensive.

Tip #6 – Be careful when you declare variables. If you don't use as something with every variable you declare, it is a variant! For example:

Code: Dim a, b, c as string. A = A variant B = A variant C = A string I've seen some people use the notation:

Code: dim x x = blah That is a NO NO! It may work yes, but its going to cost you speed.

Tip #7 – Reduce common expressions. Sometimes you have two different variables that use part of the same calculation. Instead of doing the entire calculation for both variables, eliminate the redundant calculation.

Example:

Code: x = a * b + c y = a * b + d

is slower then

Code: t = a * b x = t + c y = t + d That is especially true if your using a redundant expensive calculation in a loop.

Tip # 7 – Use long or integer for calculations. A long is a 32 bit number and is more natural on 32 bit processors. Avoid other variables such as double, single, etc

Tip #8 – Use inline functions inside of loops. Instead of calling a function, stick the code in the loop. This will make you're program larger if you repeat it in enough loops and should only be done in critical places. The reason is due to the over head of calling a function. Before the program calls a function, it has to push some things onto the stack. At the very least it will push the instruction pointer (IE: Return Address). Memory access is slow so we want to avoid that in critical places.

Tip #9 Avoid using properties in loops. Properties are accessed a lot slower then variables, so use variables instead:

Code: for i = 1 to 50 text1.text = text1.text + b(i) next i

is slower then

Code: for i = 1 to 50 strbuffer = strbuffer + b(i) next i text1.text = strbuffer

Tip #10 – Load all the data you need from the disk. Instead of loading one file at a time, load all of them at once. This will avoid future delay for the user.

Tip #11 – Make good use of the timer control. You can do background processing while waiting on a user. Use this time to prefetch data, calculations that are need, etc.

Tip #12 – Minimize dot notation in your objects! Each dot you use in a object makes visual basic do a call.

Code: Myobject.one.two.three

is slower then

Code: Myobject.one

Tip #13 Allocate enough memory at once. When you create a dynamic array and you want to add elements that haven't been allocated yet, make sure you allocate enough for all of them instead of doing it one at a time. If you don't know how many you need, times what you have allocated by 2. Allocating memory is a expensive process.

Tip #14 Avoid built in functions in loops. If you have a algorithm that is looped that requires the len of your string. Make sure you cache the size of your string in a buffer and not call the function len() with each iteration of the loop:

Code: for i = 1 to 100 sz = len(string) 'Do processing next i

instead

Code: sz = len(string) for i = 1 to 100 'Do Processing with sz next i

Tip #15 Hide the control when your setting its properties. Every time you update the properties of your control, you make it repaint. So if your developing something that displays complex graphics, may be a good idea to reduce that from happening so much.

28 Examples

This section includes a wide range of ready made code that can be easily adapted for use in your own programs. Some examples are just a few lines others are complete applications, most fall somewhere between.

Although the code is described as ready made it is not intended that you just drop it into your programs as component parts. The idea is that these examples should provide a quick start that you can build on and re-work.

Snippets[1]

Short pieces of code that illustrate techniques and solve small problems.

1 Chapter 28 on page 167

29 Snippets

29.1 TopMost Function

The code below is useful if you want to keep your application window on top or swap your application window between staying the topmost window and behaving like a standard window. Paste the code below into a code module and call either of the two routines as required.

To make your application stay on top use this call :

```
MakeTopMost Me.hwnd
```

To make your application window behave like a normal window use this call :

```
MakeNormal Me.hwnd
```

```
' Created by E.Spencer - This code is public domain.
'
Public Const HWND_TOPMOST = -1
Public Const HWND_NOTOPMOST = -2
Public Const SWP_NOMOVE = &H2
Public Const SWP_NOSIZE = &H1
Public Const SWP_NOACTIVATE = &H10
Public Const SWP_SHOWWINDOW = &H40
Public Const TOPMOST_FLAGS = SWP_NOMOVE Or SWP_NOSIZE
Public Declare Function SetWindowPos Lib "user32" _
                        (ByVal hwnd As Long, ByVal hWndInsertAfter As Long, _
                         ByVal x As Long, y, ByVal cx As Long, _
                         ByVal cy As Long, ByVal wFlags As Long) As Long

Public Sub MakeTopMost(Handle As Long)
  SetWindowPos Handle, HWND_TOPMOST, 0, 0, 0, 0, TOPMOST_FLAGS
End Sub

Public Sub MakeNormal(Handle As Long)
  SetWindowPos Handle, HWND_NOTOPMOST, 0, 0, 0, 0, TOPMOST_FLAGS
End Sub
```

29.2 Form Close Button Disable

This was posted to the misc VB news group by Ben Baird. I include it here mainly because I found it quite handy, it details the code required to disable the Form Close button (little x button at top right of the window) whilst still keeping the button visible. To test this out open a new VB project, add a command button, paste in the code below and run it.

```
Private Declare Function GetSystemMenu Lib "user32" _
```

```
        (ByVal hwnd As Long, ByVal bRevert As Long) As Long
 Private Declare Function GetMenuItemCount Lib "user32" _
        (ByVal hMenu As Long) As Long
 Private Declare Function RemoveMenu Lib "user32" _
        (ByVal hMenu As Long, ByVal nPosition As Long, ByVal wFlags As Long)
As Long
 Private Declare Function DrawMenuBar Lib "user32" _
        (ByVal hwnd As Long) As Long
 Private Const MF_BYPOSITION = &H400&
 Private Const MF_DISABLED = &H2&

 Public Sub DisableX(Frm As Form)
   Dim hMenu As Long
   Dim nCount As Long
   hMenu = GetSystemMenu(Frm.hwnd, 0)
   nCount = GetMenuItemCount(hMenu)
   Call RemoveMenu(hMenu, nCount - 1, MF_DISABLED Or MF_BYPOSITION)
   DrawMenuBar Frm.hwnd
 End Sub

 Private Sub Command1_Click()
   DisableX Me
 End Sub
```

29.3 ComboBox Automation

The code below demonstrates how to expand and hide combo box lists via code. To test it out create a new VB project, place a command button and combo box on the form and paste in the code below. When you run the project and use the tab button to move the focus from the combo box to the command button you should notice that the combo box expands and hides.

```
 Private Declare Function SendMessageLong Lib "user32" Alias "SendMessageA" _
        (ByVal hwnd As Long, _
         ByVal wMsg As Long, _
         ByVal wParam As Long, _
         ByVal lParam As Long) As Long

 Private Const CB_SHOWDROPDOWN = &H14F

 Private Sub Combo1_GotFocus()
   SendMessageLong Combo1.hwnd, CB_SHOWDROPDOWN, True, 0
 End Sub

 Private Sub Combo1_LostFocus()
   SendMessageLong Combo1.hwnd, CB_SHOWDROPDOWN, False, 0
 End Sub

 Sub Form_Load()
   Combo1.AddItem "Item 1"
   Combo1.AddItem "Item 2"
   Combo1.AddItem "Item 3"
 End Sub
```

29.4 Reversing a String

This code demonstrates a small function that reverses the content of a string. To test this out set up a form with a single command button and two text boxes then paste in the

code below. If you now enter the text "dlroW olleH" in text box 1 and press the command button you will see the reversal in text box 2, it should read "Hello World"

```
Option Explicit

Private Sub Command1_Click()
  Text2 = ReverseStr(Text1.Text)
End Sub

Private Function ReverseStr(ByVal IPStr As String) As String
  Dim i As Integer
  For i = Len(IPStr) To 1 Step -1
    ReverseStr = ReverseStr & Mid(IPStr, i, 1)
  Next
End Function
```

29.5 Preventing flicker during update

It's a common problem that controls seem to flicker as they are updated. This can be due to Windows updating a control's screen image multiple times during an update or Windows updating a control during the monitor vertical refresh. The technique below gives you the ability to lock individual controls or the entire form window during updates, this allows you to dictate to Windows when the screen updating should be done. Another way of reducing flicker is to set the form's ClipControl property to false, this forces Windows to paint the form screen as whole instead of trying to preserve the look of individual controls (it can also increase the speed of your application). For those of you having problems with flickering graphics you should investigate using the API call BitBlt (Bit Block Transfer) instead of methods like Paintpicture.

To test the code below create a new VB project and place two command buttons and a combo box on the form. The first button will populate the combo box whilst the control is locked. The second button will unlock the control and allow Windows to refresh it. Change the Hwnd to reflect the name of the control or form you want to lock.

```
 Private Declare Function LockWindowUpdate Lib "User32" (ByVal hWnd As Long) As
Long

 Private Sub Command1_Click()
   Dim i As Integer
   Combo1.Clear    ' Clear and refresh the control to show the changes
   Combo1.Refresh
   ' Lock the control to prevent redrawing
   LockWindowUpdate Combo1.hWnd
   ' Update the control
   For i = 0 To 200
     Combo1.AddItem "Entry " & i, i
   Next
   Combo1.ListIndex = 150
 End Sub

 Private Sub Command2_Click()
   ' Unlock
   LockWindowUpdate 0
 End Sub
```

29.6 Useful Date Functions

All these functions except Lastofmonth (Elliot Spener's) were sent into PCW magazine by Simon Faulkner. I've found these date functions very handy, if you have any other useful functions let me know and I'll put them on.

```
 Firstofmonth = Now() - Day(Now()) + 1

 Lastofmonth = DateAdd("m", 1, Date - Day(Date))

 Firstofyear = Now() - Datepart("y", Now()) + 1

 Lastofyear = Dateadd("yyyy", 1, Now() - Datepart("y", Now()))

 Daysinmonth = Datepart("d", Dateadd("m", 1, Now() - Day(Now())))

 Daysleftinyear = Dateadd("yyyy", 1, Now() - Datepart("y", Now())) - Now()

 Daysleftuntilchristmas = Dateadd("yyyy", 1, Now() - Datepart("y", Now())) -
Now() - 7

 Daysinyear = Dateadd("yyyy", 1, Now() - Datepart("y", Now())) - (Now() -
Datepart("y", Now()))

 Leapyear = IIf((Dateadd("yyyy", 1, Now() - Datepart("y", Now())) - (Now() -
Datepart("y", Now()))) = 366, True, False)
```

29.7 Blast Effect

Makes a circular blast effect on the picture box, make sure you rename it pic. X and Y is the center of the circle, R is the radius of the blast effect

```
 For angle=1 to 360
   pic.line (x,y) - (x + r * cos(angle*3.14159265358979/180),y + r *
sin(angle*3.14159265358979/180))
 next angle
```

29.8 Sleep Function

This is useful if you want to put your program in a wait state for a specific period of time. Just paste the code below into a new form to test it and attach it to a command button, then run it - you can view the time in the debug window. 1000 milliseconds = 1 second (but you probably knew that).

```
Private Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Private Sub Command1_Click()
  Debug.Print "Started - " & Time()
  Sleep 1000
  Debug.Print "Ended - " & Time()
End Sub
```

29.9 Random Numbers

Random numbers are not truly random if the random number generator isn't started, so you need to start it before using Rnd()

```
Randomize()
```

Replace HighestNumber and LowestNumber with your own range.

```
X=Int((HighestNumber - LowestNum + 1) * Rnd + LowestNumber)
```

29.10 Animated Mouse Cursor

The code below demonstrates how to change the mouse cursor from the base cursor to one of the animated ones. Open up a new project, add a drop list and a command button to the form then add in the code below and run it.

```
Option Explicit

Public Const GCL_HCURSOR = -12

Declare Function ClipCursor Lib "user32" _
        (lpRect As Any) _
        As Long
Declare Function DestroyCursor Lib "user32" _
        (ByVal hCursor As Any) _
        As Long
Declare Function LoadCursorFromFile Lib "user32" Alias "LoadCursorFromFileA" _
        (ByVal lpFileName As String) _
        As Long
Declare Function SetClassLong Lib "user32" Alias "SetClassLongA" _
        (ByVal hwnd As Long, ByVal nIndex As Long, ByVal dwNewLong As Long) _
        As Long
Declare Function GetClassLong Lib "user32" Alias "GetClassLongA" _
        (ByVal hwnd As Long, ByVal nIndex As Long) _
        As Long

Private mhAniCursor As Long
Private mhBaseCursor As Long
Private lresult As Long

Private Sub Command1_Click()
  ' Sort out the user selection
  If Combo1.ListIndex = 0 Then
    lresult = SetClassLong((Form1.hwnd), GCL_HCURSOR, mhBaseCursor)
    lresult = DestroyCursor(mhAniCursor)
  Else
    If Combo1.ListIndex = 1 Then
      mhAniCursor = LoadCursorFromFile("C:\windows\cursors\hourglas.ani")
    Else
      mhAniCursor = LoadCursorFromFile("C:\windows\cursors\globe.ani")
    End If
    lresult = SetClassLong((Form1.hwnd), GCL_HCURSOR, mhAniCursor)
  End If
End Sub

Private Sub Form_Load()
  ' Set up the list of cursor options
  Combo1.AddItem "Normal", 0
```

```
    Combo1.AddItem "HourGlass", 1
    Combo1.AddItem "Globe", 2
    Combo1.ListIndex = 0
    ' Grab the current base cursor
    mhBaseCursor = GetClassLong((hwnd), GCL_HCURSOR)
  End Sub
```

29.11 Adding a bitmap to a menu entry

The code below demonstrates how to add 13x13 bitmap pictures (not icons) to the left hand of each menu entry. You can define a different bitmap for both the checked and unchecked condition (as shown) or set one of these values to zero if you don't want a bitmap shown for a particular condition.

The project uses 2 picture boxes (each holding one of the required bitmaps and set to be non visible), a button and any amount of menus and submenus.

```
  Private Declare Function GetMenu Lib "user32" _
          (ByVal hwnd As Long) _
          As Long
  Private Declare Function GetSubMenu Lib "user32" _
          (ByVal hMenu As Long, ByVal nPos As Long) _
          As Long
  Private Declare Function GetMenuItemID Lib "user32" _
          (ByVal hMenu As Long, ByVal nPos As Long) _
          As Long
  Private Declare Function SetMenuItemBitmaps Lib "user32" _
          (ByVal hMenu As Long, ByVal nPosition As Long, ByVal wFlags As Long, _
           ByVal hBitmapUnchecked As Long, ByVal hBitmapChecked As Long) _
          As Long
  Private Declare Function GetMenuItemCount Lib "user32" _
          (ByVal hMenu As Long) _
          As Long

  Private Const MF_BITMAP = &H4&

  Private Sub AddIconToMenus_Click()
    Dim i1 As Long, i2 As Long, Ret As Long
    Dim MnHndl As Long
    Dim SMnHndl As Long
    Dim MCnt As Long
    Dim SMCnt As Long
    Dim SMnID As Long

    MnHndl = GetMenu(Form1.hwnd)' Get the menu handle for the current form
    MCnt = GetMenuItemCount(MnHndl)  ' Find out how many menus there are
    For i1 = 0 To MCnt - 1               ' Process each menu entry
      SMnHndl = GetSubMenu(MnHndl, i1) 'Get the next submenu handle for this
menu
      SMCnt = GetMenuItemCount(SMnHndl) 'Find out how many entries are in this
submenu
      For i2 = 0 To SMCnt - 1           'Process each submenu entry
        SMnID = GetMenuItemID(SMnHndl, i2) 'Get each entry ID for the current
submenu
        ' Add two pictures - one for checked and one for unchecked
        Ret = SetMenuItemBitmaps(MnHndl, SMnID, MF_BITMAP, Picture2.Picture,
Picture1.Picture)
      Next i2
    Next i1
  End Sub
```

29.12 Application Launching

The code below demonstrates how to launch the default "Open" action for any given file (which will normally mean launching the application that handles data files of that type). I've also included a variation of ShellExecute that allows you to launch the default system Internet browser and have it go immediately to a specified Web site.

```
 ' Required declarations
Private Declare Function ShellExecute Lib "shell32.dll" Alias "ShellExecuteA"
_
        (ByVal hwnd As Long, ByVal lpOperation As String, _
         ByVal lpFile As String, ByVal lpParameters As String, _
         ByVal lpDirectory As String, ByVal nShowCmd As Long) _
        As Long
Private Declare Function GetDesktopWindow Lib "user32" () As Long
Private Const SW_SHOWDEFAULT = 10
Private Const SW_SHOWMAXIMIZED = 3
Private Const SW_SHOWMINIMIZED = 2
Private Const SW_SHOWMINNOACTIVE = 7
Private Const SW_SHOWNA = 8
Private Const SW_SHOWNOACTIVATE = 4
Private Const SW_SHOWNORMAL = 1

Private Sub Command1_Click()
  ' Open the browser and goto a specified site
  Dim DWHdc As Long, Ret As Long
  Dim PathAndFile As String
  PathAndFile = File1.Path & "\" & File1.filename
  ' Use the desktop window as the parent
  DWHdc = GetDesktopWindow()
  Ret = ShellExecute(DWHdc, "Open", Text1.Text, "", "c:\", SW_SHOWMAXIMIZED)
End Sub

Private Sub Dir1_Change()
  File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
  Dir1.Path = Drive1.Drive
End Sub

Private Sub File1_DblClick()
  ' Launch the default "Open" action for the file
  Dim DWHdc As Long, Ret As Long
  Dim PathAndFile As String
  PathAndFile = File1.Path & "\" & File1.filename
  ' Use the desktop window as the parent
  DWHdc = GetDesktopWindow()
  Ret = ShellExecute(DWHdc, "Open", PathAndFile, "", File1.Path,
SW_SHOWNORMAL)
End Sub
```

29.13 Rounding Things Up

If you're bored of rectangular controls on rectangular forms then try the code below. Open a new project, put a command button on it, paste this code in and then run it. You should see a round button on a round form, it works on most controls. The code is fairly straightforward, you calculate the size of the ellipse required and feed this through two API calls. With a bit of playing you can get some very odd effects.

```
Private hwndDest As Long
Private Declare Function CreateEllipticRgn Lib "gdi32" _
                         (ByVal X1 As Long, ByVal Y1 As Long, _
                          ByVal X2 As Long, ByVal Y2 As Long) As Long
Private Declare Function SetWindowRgn Lib "user32" _
                         (ByVal hWnd As Long, ByVal hRgn As Long, _
                          ByVal bRedraw As Long) As Long

Private Sub Command1_Click()
  Unload Me
End Sub

Private Sub Form_Load()
  Dim hr&, dl&
  Dim usew&, useh&
  hwndDest = Me.hWnd
  usew& = Me.Width / Screen.TwipsPerPixelX
  useh& = Me.Height / Screen.TwipsPerPixelY
  hr& = CreateEllipticRgn(0, 0, usew&, useh&)
  dl& = SetWindowRgn(hwndDest, hr, True)
  hwndDest = Command1.hWnd
  usew& = Command1.Width / Screen.TwipsPerPixelX
  useh& = Command1.Height / Screen.TwipsPerPixelY
  hr& = CreateEllipticRgn(0, 0, usew&, useh&)
  dl& = SetWindowRgn(hwndDest, hr, True)
End Sub
```

29.14 TCP/Winsock - Point-to-Point Connection

This is a client-server Point-to-Point TCP over Winsock snippet, which settings are hard-coded. The snippet will connect to the server through the loopback adapter through port 50000, and the conversation would be the client sending the server a "Hello World" message which the server would show on a MsgBox. The server could only accept one connection from a client, if there is a second connection request from another client it would disconnect the first connection (thus, Point-to-Point). For a Point-to-Multipoint code (the server allows multiple connection from multiple client) see below.

29.14.1 Client Code

Add the following controls

- Winsock Control - Name="sckClient"
- Command Button - Name="Command1", Caption="Say "Hello World""
- Command Button - Name="Command2", Caption="Make a Connection"
- (Optional) Timer - Name="Timer1", Interval="1", Enabled="True"

```
   Option Explicit

   Private Sub Command1_Click()
       ' If connected, send data, if not, popup a msgbox telling to connect
first
       If sckClient.State <> sckConnected Then
           MsgBox "Connect First"
       Else
           sckClient.SendData "Hello World"
       End If
```

```
    End Sub

    Private Sub Command2_Click()
        ' If there is already a connection, close it first,
        ' failure of doing this would result in an error
        If sckClient.State <> sckClosed Then sckClient.Close

        ' OK, the winsock is free, we could open a new connection
        sckClient.Connect "127.0.0.1", 50000
    End Sub

    Private Sub Timer1_Timer()
        ' Code for seeing the status of the winsock in the form window.
        ' For the meaning of the Status Code, go to the Object Browser (F2) and
search for Winsock
        Me.Caption = sckClient.State
    End Sub
```

29.14.2 Server Code

Add the following control

- Winsock Control - Name="sckServer"
- (Optional) Timer - Name="Timer1", Interval="1", Enabled="True"

```
    Option Explicit

    Private Sub Form_Load()
        ' Listen to port 50000 for incoming connection from a client
        sckServer.LocalPort = 50000
        sckServer.Listen
    End Sub

    Private Sub sckServer_Close()
        ' If the connection is closed, restart the listening routine
        ' so other connection can be received.
        sckServer.Close
        sckServer.Listen
    End Sub

    Private Sub sckServer_ConnectionRequest(ByVal requestID As Long)
        ' If the connection is not closed close it first before accepting a
connection
        ' You can alter this behaviour, like to refuse the second connection
        If sckServer.State <> sckClosed Then sckServer.Close
        sckServer.Accept requestID
    End Sub

    Private Sub sckServer_DataArrival(ByVal bytesTotal As Long)
        Dim Data As String
        ' Receive the data (GetData),
        ' Clear the data buffer (automatic with calling GetData),
        ' Display the data on a MsgBox
        sckServer.GetData Data
        MsgBox Data
    End Sub

    Private Sub Timer1_Timer()
        ' Code for seeing the status of the winsock in the form window.
        ' For the meaning of the Status Code, go to the Object Browser (F2) and
search for Winsock
        Me.Caption = sckServer.State
    End Sub
```

29.15 TCP/Winsock - Point-to-MultiPoint Connection

This snippet is the same as the TCP/Winsock - Point-to-Point Connection above, but this code allows the server to receive multiple connection from multiple client simultaneously. This behavior is achieved by using Control Array. The Winsock Control Array index 0 is a special one, since it is never opened, it'll only listen for incoming connection, and assign to another Winsock control if there is an incoming connection. The server code is coded to reuse existing WinSock control that is already closed to receive new connection. The client code is the same as with the Point-to-Point snippet. The client will never unload Winsock control that is already open. You should understand Point-to-Point Connection before trying to implement Point-to-Multipoint connection.

29.15.1 Client Code

The same with client code for Point-to-Point Client Code[1]

29.15.2 Server Code

Add the following control

- Winsock Control - Name="sckServer", Index="0"
- (Optional) Timer - Name="Timer1", Interval="1", Enabled="True"

```
    Private Sub Form_Load()
        ' Open a listening routine on port 50000
        sckServer(0).LocalPort = 50000
        sckServer(0).Listen
    End Sub

    Private Sub sckServer_Close(Index As Integer)
        ' Close the WinSock so it could be reopened if needed
        sckServer(Index).Close
    End Sub

    Private Sub sckServer_ConnectionRequest(Index As Integer, ByVal requestID As
Long)
        Dim sock As Winsock
        ' If there is any closed Winsock, accept on that Winsock
        For Each sock In sckServer
            If sock.State = sckClosed Then
                sock.Accept requestID
                Exit Sub
            End If
        Next

        ' Else make a new Winsock
        Load sckServer(sckServer.UBound + 1)
        sckServer(sckServer.UBound).Accept requestID
    End Sub


    Private Sub sckServer_DataArrival(Index As Integer, ByVal bytesTotal As
Long)
```

1 Chapter 29.14 on page 176

```
        Dim Data As String
        ' Receive the data (GetData) for the connection that is receiving,
        ' Clear the data buffer (automatic with calling GetData) of the
receiving connection,
        ' Display the data on a MsgBox with the index of the receiving Winsock
        sckServer(Index).GetData Data, vbString
        MsgBox Data & vbCrLf & Index
    End Sub

    Private Sub Timer1_Timer()
        Dim conn As Winsock
        ' Display the status for all connection on the window bar
        ' The status code is space-separated
        Me.Caption = ""
        For Each conn In sckServer
            Me.Caption = Me.Caption & " " & conn.State
        Next
    End Sub
```

30 The Language

Many computer languages share a core set of features. Sometimes they are very similar in different languages, sometimes radically different. Some features regarded by users of one language as indispensable may be completely missing from other perfectly usable languages.

30.1 Statements

A statement in Visual Basic is like a command given to the computer. Functions and subroutines are made out of statements.

A statement generally consists of the name of a subroutine followed by a list of arguments:

```
Debug.Print "Hello World"
```

The first word (Debug) is the name of a built in object in Visual Basic. the second word (Print) is the name of a method of that object. "Hello World" is the argument to the method. The result is:

```
Hello World
```

printed to the *Immediate Window*.

In Visual Basic Classic subroutine call statements can be written in two ways:

```
x "Hello", "World"
Call x("Hello", "World")
```

x is a subroutine which takes two string arguments. The keyword *Call* is a leftover from older versions of Basic. If you use this form you must enclose the entire argument list in a parenthesis. Sometimes the programmer wants to call a function but is not interested in the return value only in the side-effects. In such cases he or she will call the function as though it were a subroutine, however only the first form above can be used.

There are other more complex statements which can span several lines of code:

- If..Else..End If
- Do While..Loop

These compound statements include other statements and expressions.

30.2 Variables

Like most programming languages, Visual Basic is able to use and process named variables and their contents. **Variables** are most simply described as names by which we refer to some location in memory - a location that holds a value with which we are working.

It often helps to think of variables as a 'pigeonhole', or a placeholder for a value. You can think of a variable as being equivalent to its value. So, if you have a variable i that is initialized to 4, $i+1$ will equal 5.

In Visual Basic variables can be *declared* before using them or the programmer can let the compiler allocate space for them. For any but the simplest programs it is better to declare all variables. This has several benefits:

- you get the datatype you want instead of *Variant*,
- the compiler can sometimes check that you have used the correct type.

All variables in Visual Basic are typed, even those declared as *Variant*. If you don't specify the type then the compiler will use Variant. A Variant variable is one that can hold data of any type.

30.2.1 Declaration of Variables

Here is an example of declaring an integer, which we've called `some_number`.

```
Dim some_number As Integer
```

You can also declare multiple variables with one statement:

```
Dim anumber, anothernumber, yetanothernumber As Integer
```

This is not a good idea because it is too easy to forget that in Visual Basic the type name applies only to the *immediately* preceding*variable name. The example declares two*Variants*and* one *Integer*

You can demonstrate what actually happens by this little program:

```
Option Explicit

Public Sub main()

  Dim anumber, anothernumber, yetanothernumber As Integer
  Dim v As Variant

  Debug.Print TypeName(anumber), VarType(anumber), _
       TypeName(anothernumber), VarType(anothernumber), _
       TypeName(yetanothernumber), VarType(yetanothernumber)

  Debug.Print TypeName(anumber), VarType(v)

  Debug.Print "Assign a number to anumber"
  anumber = 1
  Debug.Print TypeName(anumber), VarType(anumber)
```

```
    Debug.Print "Assign a string to anumber"
    anumber = "a string"
    Debug.Print TypeName(anumber), VarType(anumber)

End Sub
```

The result is:

```
Empty           0           Empty           0           Integer        2
Empty           0
Assign a number to anumber
Integer         2
Assign a string to anumber
String          8
```

Notice that both VarType and TypeName return information about the variable stored inside the Variant not the Variant itself.

To declare three integers in succession then, use:

```
Dim anumber As Integer, anothernumber As Integer, yetanothernumber As Integer
```

In Visual Basic it is not possible to combine *declaration* and *initialization* with variables (as VB does not support constructors - see Objects and Initialise though), the following statement is illegal:

```
Dim some_number As Integer = 3
```

Only constant declarations allow you to do this:

```
Const some_number As Integer = 3
```

In Visual Basic variables may be declared anywhere inside a subroutine, function or property, where their scope is restricted to that routine. They may also be declared in the *declarations* section of a module before any subroutines, functions or properties, where they have module level scope.

By default, variables do not have to be declared before use. Any variables created this way, will be of *Variant* type. This behaviour can be overrriden by using the Option Explicit statement at the top of a module and will force the programmer to declare all variables before use.

After declaring variables, you can assign a value to a variable later on using a statement like this:

```
Let some_number = 3
```

The use of the Let keyword is optional, so:

```
some_number = 3
```

is also a valid statement.

You can also assign a variable the value of another variable, like so:

```
anumber = anothernumber
```

In Visual Basic you cannot assign multiple variables the same value with one statement. the following statement does not do what a C programmer would expect:

```
anumber = anothernumber = yetanothernumber = 3
```

Instead it treats all but the first = signs as *equality test* operators*and then assigns either*True*or*False*to anumber* depending on the values of *anothernumber* and *yetanothernumber*.

Naming Variables

Variable names in Visual Basic are made up of letters (upper and lower case) and digits. The underscore character, "_", is also permitted. Names must not begin with a digit. Names can be as long as you like.

Some examples of valid (but not very descriptive) Visual Basic variable names:

```
foo
Bar
BAZ
foo_bar
a_foo42_
QuUx
```

Some examples of invalid Visual Basic variable names:

2foo

must not begin with a digit

my foo

spaces not allowed in names

$foo

$ not allowed -- only letters, digits, and _

while

language keywords cannot be used as names

_xxx

leading underscore not allowed.

Certain words are reserved as keywords in the language, and these cannot be used as variable names, or indeed as names of anything else.

In addition there are certain sets of names that, while not language keywords, are reserved for one reason or another.

The naming rules for variables also apply to other language constructs such as function names and module names.

It is good practice to get into the habit of using descriptive variable names. If you look at a section of code many months after you created it and see the variables `var1`, `var2` etc., you will more than likely have no idea what they control. Variable names such as `textinput` or `time_start` are much more descriptive - you know exactly what they do. If you have a variable name comprised of more than one word, it is convention to begin the next word with a capital letter. Remember you can't use spaces in variable names, and sensible capital use makes it easier to distinguish between the words. When typing the variable name later in the code, you can type all in lowercase - once you leave the line in question, if the variable name is valid VB will capitalize it as appropriate.

30.2.2 Literals

Anytime within a program in which you specify a value explicitly instead of referring to a variable or some other form of data, that value is referred to as a **literal**. In the initialization example above, 3 is a literal. Numbers can be written as whole numbers, decimal fractions, 'scientific' notation or hex. To identify a hex number, simply prefix the number with &H.

30.3 Conditions

Conditional clauses are blocks of code that will only execute if a particular expression (the condition) is true.

30.3.1 If Statements

If statements are the most flexible conditional statements: see Branching[1]

30.3.2 Select Case

Often it is necessary to compare one specific variable against several values. For this kind of conditional expression the *Select Case*[2] statement is used.

30.4 Unconditionals

Unconditionals[3] let you change the flow of your program without a condition. These are the Exit, End and Goto statements.

1 Chapter 4.4 on page 32
2 Chapter 5.3 on page 35
3 Chapter 5.4 on page 35

30.5 Loops

Loops[4] allow you to have a set of statements repeated over and over again.

To repeat a given piece of code a set number of times or for every element in a list: see For..Next Loops[5]

4 Chapter 6.2 on page 40
5 Chapter 6.1 on page 39

31 Coding Standards

See ../Credits and Permissions/[1] for details of copyright, licence and authors of this piece.

Work in progress. Document needs substantial reformatting to fit the Wikibooks style. Also the page is much too long for convenient editing in a browser text box so it must be broken into sections. Use the original as a guide: `http://www.gui.com.au/resources/coding_standards_print.htm`

31.1 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 acknowledgement 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 we have simply had to make a choice between two or more equally valid alternatives. We have tried to actually think about the relative merits of each alternative but inevitably some personal preferences have come into play.

Hopefully this document is actually readable. Some standards documents 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 should be treated any less lightly than its drier cousins.

31.2 When Does This Document Apply?

It is the intention that all code 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 a code library that already uses this standard.

Code written for customers that require that their standards be adopted.

1 `http://en.wikibooks.org/wiki/..%2FCredits%20and%20Permissions%2F`

It is not uncommon for programmers 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.

31.3 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.

31.3.1 CamelCasing and uppercasing

In a nutshell variable name body, function names, and label names are using **CamelCasing**, while constants use **UPPER_CASING**.

Roughly speaking, camel casing is that all words have the 1st letter in uppercase without any word separation, and the subsequent letters in lowercase, while uppercasing is that all words are in upper case with an underscore separator.

31.3.2 Name body casing does not apply to prefixes and suffixes

Prefixes and suffixes are used for

- module name for name with module scope
- variable type
- whether a variable is a function parameter

Those prefixes and suffixes are not subject to the same casing as the name body. Here are a few example with the prefixes or suffixes in bold face.

```
' s shows that sSomeString is of type String
sSomeString
' _IN shows that iSomeString_IN is an input argument
iSomeInteger_IN
' MYPRJ_ shows that MYPRJ_bSomeBoolean has module scope MYPRJ
MYPRJ_bSomeBoolean
```

31.3.3 Physical units should not be cased

Sometimes it is convenient to show within a variable name in which physical unit the variable value is expressed, like mV for milliVolt, or kg for kilogramme. The same applies to a function name returning some real number value. The physical unit should be placed at the end with a leading underscore and without any casing. Case has a special meaning for physical units, and it would be quite disturbing to change it. Here are a few examples where the unit and underscore are in bold face.

```
fltSomeVoltage_mV
sngSomePower_W_IN
dblSomePowerOffset_dB
dblSomeSpeed_mps
```

31.3.4 Acronyms

Acronym words have special handling.

Examples			
Free text	**Camel Casing**	**Upper casing**	**Comment**
Am I Happy	AmI_Happy	AM_I_HAPPY	One letter words are all upper case, and are followed by an underscore inCamel casing
The GSM phone	TheGSMPhone	THE_GSm_PHONE	Acronyms are kept all upper case in Camel casing, and have last letterset lower case with upper-casing
A DC-DC Converter	A_DC_DCConverter	A_Dc_Dc_CONVERTER	When there are two consecutive acronyms, and the first acronym has only twoletters, then in Camel casing, the first acronym is followed by an underscore

Examples			
Free text	**Camel Casing**	**Upper casing**	**Comment**
A GSM LTE phone	A_GSmLTEPhone	A_GSm_LTe_PHONE	When there are two consecutive acronyms, and the first acronym has three or moreletters, then in Camel casing, the first acronym has last letter in lower case

31.4 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.

31.5 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.

31.6 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.

31.7 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.

31.8 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.

31.9 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.

31.10 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.

31.11 Procedure

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.

31.12 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.

31.13 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
```

31.14 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, one limitation imposed by this scheme. Which 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.

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. You can also read the value of the function return variable.

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.

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

31.15 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
```

@TODO: Global versus Public @TODO: Mention enums.

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.

31.16 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 "Labeln" 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.)

31.17 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".

31.18 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
```

31.19 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)
```

31.20 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.

31.21 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

31.22 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 a 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
```

31.23 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.

31.24 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.

@TODO: rewrite the short name specific parts, update to win32.

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.

Each programmer should 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.

31.25 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.

31.26 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.

31.27 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.

31.28 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.

31.29 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.

31.30 Assignment Statements (Let)

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

31.31 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.

31.32 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 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
End Sub
```

31.33 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.

31.34 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, this 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).

31.35 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.

This standard deprecates the use of 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.

31.36 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
```

31.37 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 a 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 HandleError
  . . .
Done:
  Exit Sub
```

```
HandleError:
  Resume Done
End Sub
```

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 HandleError
```

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.

31.37.1 Standard Error Handler

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.

Using the standard error handler involves the following steps:

- add the module containing the defalt error handler 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.

The default error handler can also provide exits in which you can further customise the way the generic error handler works without modifying the code. These would take the form of global procedures 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.

Here are some examples of constants and procedures you may want to implement:

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)

31.38 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.

31.39 White Space and Indentation

31.39.1 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.

31.39.2 White space in expressions

Put one space around operators, and after a comma in expressions.

```
Let miSomeInteger = 1 + 2 * (miSomeOtherInteger - 6)
```

31.40 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.

31.40.1 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.

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.

31.40.2 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.

31.40.3 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?

32 VB6 Command Reference

This page aims to be a comprehensive command reference to the MS Visual Basic 6 programming language.

32.1 String Manipulation

32.1.1 Asc

Returns the number corresponding to the ASCII code for the first character of string

Usage

```
Asc(string)
```

string = string containing character as first letter whose ASCII code is to be found.

Example

```
code = Asc("apple")
```

Here `code` will get value 97

32.1.2 Chr

Returns a string character that corresponds to the ASCII code in the range 0-255. Reverse of the `Asc` function.

Usage

```
Chr(code)
```

code = ASCII code.

Example

```
char = Chr(97)
```

Here `char` gets value "a".

32.1.3 Len

Returns the length of a given string or 0 for Empty.

Usage

```
Len(expression)
```

expression = a string or Empty

Example

```
mystring = InputBox("Enter a string to test")

length = Len(mystring)

MsgBox "Length of the string is " & length
```

e.g. where mystring is "Hello", length will be 5.

32.1.4 Left

Returns a given number of characters from the left hand side of a string

Usage

```
Left(string,x)
```

string = string to use

x = number of characters

Example

```
mystring = InputBox("Enter a string")

mystring = Left(mystring, 4)

MsgBox "First four characters of your input are " + mystring
```

e.g. where the input mystring is "Hello", the output mystring will be "Hell"

32.1.5 Right

Returns a given number of characters from the right hand side of a string

Usage

```
Right(string, x)
```

string = string to use

x = number of characters

Example

```
mystring = InputBox("Enter a string")

mystring = Right(mystring, 4)

MsgBox "Last four characters of your input are " + mystring
```

e.g. where the input mystring is "Hello", the output mystring will be "ello"

32.1.6 Mid (Function)

Returns a given number of characters from the middle of a string

Usage

```
Mid(string, start, length)
```

string = string to use

start = character to start at (1 is the first character)

length = number of characters

Example

```
mystring = InputBox("Enter a string")

mystring = Mid(mystring, 2, 3)

MsgBox "The second, third, and fourth characters of your input are " & mystring
```

e.g. where the input mystring is "Hello", the output mystring will be "ell"

32.1.7 Mid (Statement)

Sets a given number of characters in the middle of a string equal to the same number of characters from the beginning of another string

Usage

```
Mid(mystring, start, length)
```

mystring = the string to take characters from start = character to start at (1 is the first character)

length = number of characters

Example

```
mystring = InputBox("Enter a string")

Mid(mystring, 2, 3) = "abcd"

MsgBox "Your string with abc as the second, third, and fourth characters of
your input are " + mystring
```

e.g. where the input mystring is "Hello", the output mystring will be "Habco"

32.1.8 Trim

Removes leading and trailing spaces from a string

Usage

```
Trim(string)
```

string = string to use

Example

```
mystring = Trim(mystring)
```

e.g. where the original value of mystring was " Hello ", the new value of mystring will be "Hello".

32.1.9 LCase

Converts a string to lowercase

Usage

```
LCase(string)
```

string = string to use

Example

```
mystring = LCase(mystring)
```

e.g. where the original value of mystring was "HELLO", the new value of mystring will be "hello".

32.1.10 UCase

Converts a string to uppercase

Usage

```
UCase(string)
```

string = string to use OR character

Example

```
mystring = UCase(mystring)
```

e.g. where the original value of mystring was "Hello", the new value of mystring will be "HELLO".

32.1.11 String

Creates a string with the specified length of the specified character

Usage

```
String(length, character)
```

length = length of string

character = character to fill string with

Example

```
mystring = String(5,"a")
```

e.g. the new value of mystring will be "aaaaa".

32.1.12 Space

Creates a string with the specified length of space

Usage

```
Space(length)
```

length = length of string

Example

```
mystring = Space(5)
```

e.g. the new value of mystring will be "   ".

32.1.13 StrConv

Returns a converted string as specified.

Usage

```
StrConv(string, conversion,LCID)
```

string = string to use

conversion = case to convert the sting to (lowercase: vbLowerCase, uppercase: vbUpperCase, proper case (first letter in caps): vbProperCase) LCID = optional. The LocaleID, if different than the system LocaleID.

Example

```
mystring = StrConv(mystring, vbProperCase)
```

e.g. where the original value of mystring was "HELLO", the new value of mystring will be "Hello".

32.2 Mathematical Functions

32.2.1 Abs

Returns the absolute value of a number

Usage

Abs(number)

Example

msgbox "The absolute value of -4 is " & abs(-4)

Important note!

The Abs function only accepts numbers. Non-numeric values generate an error.

32.2.2 Cos

Returns the cosine of a number

Usage

Cos(integer)

Example

msgbox "The cosine of 4 is " & cos(4)

Important note!

The input angle for all VB trigometric functions should be in radians[1]. To convert degrees to radians, multiply degrees by pi / 180.

32.2.3 Sin

Returns the sine of a number

Usage

1 http://en.wikipedia.org/wiki/radian

Sin(integer)

Example

msgbox "The sine of 4 is " & sin(4)

Important note!

The input angle for all VB trigometric functions should be in radians. To convert degrees to radians, multiply degrees by pi / 180.

32.2.4 Tan

Returns the tangent of an angle

Usage

Tan(integer)

Example

msgbox "The tangent of 4 is " & tan(4)

Important note!

The input angle for all VB trigometric functions should be in radians. To convert degrees to radians, multiply degrees by pi / 180.

32.2.5 Val

Returns the numeric value of a string

Uses "." as decimal separator. (It does not depend on the Regional Settings)

Usage

Val(string)

Example

```
Val("5")
```

The return value will be 5

```
Val("10.6")
```

The return value will be 10.6

```
Val("Hello World")
```

The return value will be 0 (zero)

```
txtOutput.Text = Val(txtNumber1.Text) * Val(txtNumber2.Text)
```

The text box txtOutput will contain the product of the numbers in the txtNumber1 and txtNumber2 text boxes. If either of the text boxes contains a value that cannot be evaluated to a number then the output will be 0 (zero).

32.2.6 Rnd

Returns a floating point number less than 1 but greater than or equal to 0.

Usage

```
Rnd[(number)]
```

You can change the seed the random number generator uses by providing a value as a parameter e.g. Rnd(500) will use the number 500 as the seed in generating random numbers.

- If you provide a value less than zero then the same number will be returned by the Rnd() function
- A value of zero will return the most recently generated random number.
- Providing no value will generate the next random number in the sequence.
- Providing a positive number will generate the next random number in the sequence using *number* as a seed.

Example

```
myvar = Rnd() * 10
```

myvar will become a random number between 0 and 10.

```
myvar = (Rnd() * 15) + 25
```

myvar will become a random number between 25 and 40.

```
myvar = (Rnd(12345) * 50) - 25
```

myvar will become a random number between -25 and 25 and 12345 will be used as the seed to generate this number.

32.2.7 Int

Returns the integer portion of a number.

Usage

Int(n)

Where n is the number to return

Example

```
Int(10)
```

Returns 10

```
Int(5.1)
```

Returns 5

```
Int(5.86)
```

Returns 5

```
MsgBox "The integer portion of " & myvar & " is " & Int(myvar) & "."
```

NOTE: No rounding will be performed, the decimal portion of the number will simply be removed. If you want to round a number to the nearest whole integer then use CInt. One interesting point to remember about CInt is that if the decimal portion of the number is exactly .5 then it will be rounded down if the number is odd and rounded up if it is even. For example CInt(1.5) and CInt(2.5) will both return 2.

32.3 Input and output

32.3.1 MsgBox

Displays a message box

Usage

MsgBox prompt,buttons,title,helpfile,context

where prompt = the text displayed on the message box buttons = combination of keywords to set out the layout of buttons and icons on the message box title = the title displayed in the message box titlebar

Where the output from the message box is being assigned to a value (such as when there is a choice to be made, use parenthesis thus:

string=MsgBox (prompt,buttons,title,helpfile,context)

If no title is defined, the program name will be used instead. Button combination defaults to vbOKOnly if none is defined, and no icon will be displayed.

Button Combinations

vbOKOnly vbOKCancel vbAbortRetryIgnore vbYesNoCancel vbYesNo vbRetryCancel

Icon Display

vbCritical vbQuestion vbExclamation vbInformation

Default Button

vbDefaultButton1 vbDefaultButton2 vbDefaultButton3 vbDefaultButton4

Examples

MsgBox "An error has occurred!",vbExclamation,"Error"

Note that since the message is fixed, quotation marks have been used.

Response=MsgBox("Yes or no?",vbYesNo + vbQuestion,"Choose one")

Note the parenthesis and how the vbYesNo and vbQuestion constants have been combined

MsgBox "An error has occurred on line " & lineno, vbExclamation, "Error"

32.3.2 InputBox

Displays a simple input box for user to enter data

Usage

Variable=InputBox(Prompt,Title)

Variable = variable to which the input value will be stored Prompt = text displayed to the user in the input box Title = title of the input box

Examples

myint=InputBox("Enter a number","Enter a number")

32.4 Date and Time

32.4.1 Date

Returns or sets the current system date

Example

MsgBox "Today 's date is " & Date & "."

32.4.2 Time

Returns or sets the current system time

Example

MsgBox "The time now is " & Time & "."

Sidenote: be wary of changing date and time in your program. Many other programs will depend upon the system's date and time being set correctly, so altering it could have a knock on effect. Use date and time changes sparingly, and notify the user if this is being done.

32.4.3 Timer

Returns the number of seconds since midnight

Example

NumHrs=Int(Timer / 3600) MsgBox "Currently in hour " & NumHrs

32.4.4 WeekdayName

Returns the name of a day of the week

Usage

WeekdayName(x,FirstDayOfWeek)

x = integer between 1 and 7 FirstDayOfWeek (OPTIONAL) = the day counted as the first in the week. The default is Sunday.

Example

DayName=WeekdayName(i,vbMonday)

32.5 Miscellaneous

32.5.1 IsNull

Returns true if an expression is null and false otherwise.

Example:

```
If IsNull(vntMyVariantVar) Then
    Debug.Print "vntMyVariantVar is null"
Else
    Debug.Print "vntMyVariantVar is NOT null"
End If
```

32.6 File Handling

32.6.1 Open

Open "" & namefile& ".dat" For Binary As #ff

32.7 Selection

If... Then... Else... Performs selection based upon criteria. A cornerstone of any VB program.

'Usage'

Single Line Selection

```
If condition Then action
```

Note that only one line can be used with this style of statement

Multi Line Selection

```
If condition Then

[code to execute if condition is true]

EndIf
```

Using Else

```
If condition Then

[code to execute if condition is true]

Else

[code to execute if condition is false]

EndIf
```

Condition is a condition which must be met for the code within the statement to be executed. It can make use of the modifiers AND, NOT, OR and XOR (e.g. x=4 AND y > 6). See also the Examples section.

32.7.1 Examples

Simple one-line selection:

```
If x < 10 then MsgBox "The number is below 10"
```

Slightly more complex one-line selection, using the AND modifier to check two conditions at one time.

```
If x > 10 AND x < 20 then MsgBox "The number is between 10 and 20"
```

Simple multi-line selection. The code between the If... and EndIf statements will be executed only if the condition is met.

```
If Date = 25/12/2006 Then
    MsgBox "The date is Christmas Day"
    Call ChristmasDay
EndIf
```

Using the Else statement. If the variable i is over 4, the program will exit the subroutine. Otherwise, it will go to the label SaveFile.

```
If i > 4 Then
    Exit Sub
Else
    Goto SaveFile
EndIf
```

Two If... Endif statements are used, one nested within the other. If i is 6 or 7, the code between the inside If...Endif statements is executed. Then if i=7 the first message box will be shown; otherwise (i.e. if i=6) the other will be displayed. For both cases, the final message box will be shown. When nesting selection statements, it's often a good idea to indent them as shown here.

```
If i = 6 OR i = 7 Then
   If i = 7 Then
       MsgBox "Today is Sunday"
   Else
       MsgBox "Today is Saturday"
   EndIf
   MsgBox "Today is the Weekend"
EndIf
```

33 Glossary

An alphabetical list of definitions crossreferenced to more thorough explanations in the main text or in external documents.

33.1 A

argument

In computing jargon an *argument* is one of the pieces of data passed to a procedure[1]. Another name is *parameter*.

assign, assigning, assignment

Assignment is one of the fundamental operations of computing. All it means is copying a *value* into the *memory location* pointed at by a *variable*. The value can be a *literal* or the value of some other *variable*, q.v. Assignment[2]

33.2 B

ByRef

Declares an argument to a procedure[3] as a pointer[4] to the argument[5] instead of as a copy of the *value* of the argument. This allows the procedure to permanently change the variable. If neither ByRef or ByVal is stated, ByRef is assumed by the compiler.

ByVal

Declares an argument to a procedure[6] as a copy of the *value* of the argument. The value of the original variable cannot be changed by the procedure. The value of the newly created variable can be changed within the procedure, but this does not affect the variable it was copied from. Programs are more robust if variables are declared ByVal whenever possible since this means that an argument will not be unexpectedly changed by calling a function. This also results in a faster program (see pg 758 of the Microsoft Visual Basic 6 Programmer's Guide).

---

1 Chapter 33.9 on page 237
2 http://en.wikipedia.org/wiki/Assignment%20%28computer%20science%29
3 Chapter 33.9 on page 237
4 Chapter 33.9 on page 237
5 Chapter 33.9 on page 237
6 Chapter 33.9 on page 237

33.3 C

compiler directives[7]

These are instructions included in the text of the program that affect the way the compiler behaves. For instance it might be directed to include one or another version of a piece of code depending on whether the target operating system is Windows 95 or Windows XP.

33.4 I

Immediate Windows

This is the window in the IDE which receives output from *Debug.Print*. See ../IDE/[8]

33.5 J

JavaScript

JavaScript is an interpreted language mostly used to provide scripts inside web pages. It is, however, not confined to such uses. See Jarithmetic[9], a case study in multilingual programming and Crockford on JavaScript[10], the world's most misundertood language.

JScript

Microsoft's implementation of JavaScript as provided by the Windows Script Host. See Jarithmetic[11] for an example of its use.

33.6 O

Operands and Operators

An *operand* is operated on by an *operator*. *Expressions* are built of *operands* and *operators*. For instance in this expression:

```
a = b + c
```

there are three operands (a, b, and c) and two operators (= and +).

Operands in Visual Basic Classic are either variables[12] or literals[13]

7 http://en.wikibooks.org/wiki/..%2FCompiler%20Directives
8 http://en.wikibooks.org/wiki/..%2FIDE%2F
9 http://en.wikibooks.org/wiki/..%2FJArithmetic
10 http://www.crockford.com/javascript/
11 http://en.wikibooks.org/wiki/..%2FJArithmetic
12 Chapter 33.9 on page 237
13 Chapter 33.9 on page 237

33.7 P

procedure

A subroutine[14], function[15], method[16], or property[17]

33.8 R

ragged array

An array having rows of differing lengths. Such arrays are natural in languages such as C and JavaScript but rather unusual in Visual Basic although there is nothing that prevents their creation. In Visual Basic you can create such an array using the *Array* function:

```
aRagged = Array(Array(1, 2, 3), Array(1, 2))
```

Such arrays are inefficient in Visual Basic because they are implemented as *Variants* but functionaly identical arrays can be created as instances of a class and can be made much more efficient both in storage space and execution time, although not quite as efficient as *C* arrays.

real number

A variable that can hold a *real number* is one that can hold a number that can have any value including fractional value. In computer languages variables only approximate real numbers because the irrational numbers are also real, see Real number[18]. In Visual Basic Classic there are two real number types: Single and Double, see ../Data Types/[19]

reference

A variable that holds a *pointer* to a *value* rather than holding the the *value* itself. In strict Visual Basic usage only *object references* work like this.

reference argument

A declaration of an argument using the ByRef[20] keyword. *Reference arguments* allow the subroutine or function to make changes to the variable in the calling routine. See ../Procedures and Functions/[21].

33.9 S

subroutine

14 Chapter 33.9 on page 237
15 Chapter 33.9 on page 237
16 Chapter 33.9 on page 237
17 Chapter 33.9 on page 237
18 http://en.wikipedia.org/wiki/Real%20number
19 Chapter 12 on page 72
20 Chapter 33.9 on page 237
21 Chapter 13.7 on page 78

A procedure similar to a function but it does not return a value.

34 Work in Progress

This is a work area. Short pieces of text, reminders and links that don't warrant a page to themselves but need a temporary home lest they be forgotten start life here.

34.1 Miscellaneous

- http://www.elitevb.com/content/01,0057,01/01.aspx
- Split Effective Programming into separate pages as it is now over 36K
- Explain file format of VBP, frm, cls, bas, etc.
- Registry entries that refer to VB components, how they are structured, what you can do with them.
- Side by side installation.
- Windows standard dialogs http://builder.com.com/5100-6371_14-5800282.html, http://www.vbaccelerator.com/home/VB/Code/Libraries/Common_Dialogs/Folder_Browser/article.asp, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnima01/html/ima0201.asp, http://www.devx.com/vb2themax/Tip/19255,

34.2 Graphics

- http://www.a1vbcode.com/app-2571.asp

34.3 Unicode Support

Unicode Issues with VB , Add Unicode Support, Unicode Custom Controls,

34.4 Networks

inet control winsock control comm control

34.5 Shell Integration

34.6 Shell Context Menu

Add examples for VBZip etc.

34.7 Command Line

34.8 Console Programs

See:

- http://www.nirsoft.net/vb/console_application_visual_basic.html
- http://www.flexbeta.net/forums/lofiversion/index.php/t3090.html

A very simple way of creating console programs: http://www.kellyethridge.com/vbcorlib/doc/Console.html

34.9 CGI

See:

- http://www.vbcity.com/forums/faq.asp?fid=10&cat=Web+Development&

34.10 Windows management Interface

http://archive.is/20130628221511/http://techrepublic.com.com/5208-11196-0.html?forumID=44&threadID=121079

34.11 Sound

- http://www.unh.edu/music/Visual_Basic_Programs/Visual_Basic.htm
- http://www.devx.com/vb2themax/Tip/18323
- http://www.ostrosoft.com/vb/projects/mci_play/index.asp
- http://vbcity.com/forums/topic.asp?tid=103678
- http://www.xtremevbtalk.com/archive/index.php/t-237143.html
- http://www.vbforums.com/showthread.php?t=359612
- http://www.vbaccelerator.com/home/VB/Code/vbMedia/Audio/Lossless_WAV_Compression/article.asp

35 Links

35.1 External Links

Link	Comment
Visual Basic Tutorial[1]	37 Lessons on Visual Basic 6 and sample source code
Visual Basic 6 (VB6) tutorials and source code[2]	Example Visual Basic 6.0 tutorials and source code
VBCorLib[3]	Do exceptions in Visual Basic 6.0
Visual Basic Tutorials[4]	
WinsockVB[5]	Everything there is to know about sockets and Visual Basic 6.0. Still has active forums. *OFF LINE*
VBSpeed[6]	Objective evidence you can use to help you write faster code
GPWiki[7]	Contains more game related Visual Basic tutorials
XtremeVBTalk[8]	A VB forum
AllAPI[9]	Examples for Windows APIs
Hardcore Visual Basic[10]	Bruce McKinney's masterpiece; shows that you can write almost anything in Visual Basic 6.0.
Planet Source Code[11]	Sample Source Code
VBCode.com[12]	Free Source Code for VB
A1VBcode.com[13]	*Another* Free Source Code site for VB
VBexemple.com[14]	Free examples(source code), Tutorials for Visual Basic 6.0

1 http://www.vbtutor.net
2 http://www.vb6.us
3 http://www.kellyethridge.com/vbcorlib/index.shtml
4 http://www.pickatutorial.com/tutorials/vb_1.htm
5 http://www.winsockvb.com/
6 http://xbeat.net/vbspeed
7 http://gpwiki.org
8 http://www.visualbasicforum.com/
9 http://www.mentalis.org/vbexamples/list.php?category=MISC
10 http://vb.mvps.org/hardcore/
11 http://www.planet-source-code.com
12 http://www.vbcode.com/
13 http://www.a1vbcode.com/
14 http://www.vbexemple.com/

Link	Comment
Visual Basic Beginners Tutorials[15]	Tutorials for Visual Basic.NET

15 http://www.visualbasictutorial.net/

36 Contributors

36.1 Authors and Contributors

Details of permissions given to include content from other sites and documents is on the Credits and Permissions[1] page.

Name	Comment
Batjew[2]	Wrote almost everything to start, getting help now.
EugeneH[3]	Corrected mistakes and wrote the rest. User name is now Orage[4]
Kwhitefoot[5]	Added: distinction between Type and Class, optimization of programs. Split into separate chapters. Added coding standards, case studies, optimization.
Elliot Spencer	Contributed many examples from his web site: `http://www.ilook.fsnet.co.uk/index.htm`
Aadnk[6]	Some sections translated from the Norwegian Visual Basic[7] WikiBook (by Kwhitefoot[8]). Also wrote the Windows Dialogs chapter.
GUI Computing Pty Ltd/Mark Trescowthick[9]	Mark Trescowthick kindly gave permission for the `http://www.gui.com.au/resources/coding_standards.htm` GUI Computing coding standards[10] and `http://www.avdf.com/` Australian Visual Developers Forum[11] to be used, see Credits and Permissions[12].

1 http://en.wikibooks.org/wiki/..%2FCredits%20and%20Permissions
2 http://en.wikibooks.org/wiki/User%3ABatjew
3 http://en.wikibooks.org/wiki/User%3AEugeneH
4 http://en.wikibooks.org/wiki/user%3Aorage
5 http://en.wikibooks.org/wiki/User%3AKwhitefoot
6 http://no.wikibooks.org/wiki/User:Aadnk
7 http://no.wikibooks.org/wiki/Visual_Basic
8 http://en.wikibooks.org/wiki/User%3AKwhitefoot
9 http://www.gui.com.au
10 http://en.wikibooks.org/wiki/%20GUI%20Computing%20coding%20standards
11 http://en.wikibooks.org/wiki/%20Australian%20Visual%20Developers%20Forum
12 http://en.wikibooks.org/wiki/..%2FCredits%20and%20Permissions

37 Contributors

Edits	User
2	A R King[1]
71	Adrignola[2]
2	Albmont[3]
2	Alexius08[4]
1	Avicennasis[5]
3	Az1568[6]
1	Bevo[7]
3	Bijee[8]
1	Carl Turner[9]
55	Dan Polansky[10]
2	Darklama[11]
1	Derbeth[12]
3	Gang65[13]
4	Hagindaz[14]
1	HethrirBot[15]
1	JackPotte[16]
18	Jguk[17]
4	Jomegat[18]
1	Jorgenev[19]
1	King jakob c 2[20]
2	Krischik[21]

1 http://en.wikibooks.org/wiki/User:A_R_King
2 http://en.wikibooks.org/wiki/User:Adrignola
3 http://en.wikibooks.org/wiki/User:Albmont
4 http://en.wikibooks.org/wiki/User:Alexius08
5 http://en.wikibooks.org/wiki/User:Avicennasis
6 http://en.wikibooks.org/wiki/User:Az1568
7 http://en.wikibooks.org/wiki/User:Bevo
8 http://en.wikibooks.org/wiki/User:Bijee
9 http://en.wikibooks.org/wiki/User:Carl_Turner
10 http://en.wikibooks.org/wiki/User:Dan_Polansky
11 http://en.wikibooks.org/wiki/User:Darklama
12 http://en.wikibooks.org/wiki/User:Derbeth
13 http://en.wikibooks.org/wiki/User:Gang65
14 http://en.wikibooks.org/wiki/User:Hagindaz
15 http://en.wikibooks.org/wiki/User:HethrirBot
16 http://en.wikibooks.org/wiki/User:JackPotte
17 http://en.wikibooks.org/wiki/User:Jguk
18 http://en.wikibooks.org/wiki/User:Jomegat
19 http://en.wikibooks.org/wiki/User:Jorgenev
20 http://en.wikibooks.org/wiki/User:King_jakob_c_2
21 http://en.wikibooks.org/wiki/User:Krischik

361	Kwhitefoot[22]
2	LlamaAl[23]
3	Mike.lifeguard[24]
1	Mts15[25]
13	Orage[26]
6	Panic2k4[27]
1	Pmlineditor[28]
5	Psoup[29]
7	QuiteUnusual[30]
1	Ramac[31]
6	Recent Runes[32]
1	Red4tribe[33]
31	Redrocketboy[34]
4	Reece[35]
1	Sigma 7[36]
1	Taxman[37]
1	Thenub314[38]
2	Webaware[39]
1	Xania[40]

22 http://en.wikibooks.org/wiki/User:Kwhitefoot
23 http://en.wikibooks.org/wiki/User:LlamaAl
24 http://en.wikibooks.org/wiki/User:Mike.lifeguard
25 http://en.wikibooks.org/wiki/User:Mts15
26 http://en.wikibooks.org/wiki/User:Orage
27 http://en.wikibooks.org/wiki/User:Panic2k4
28 http://en.wikibooks.org/wiki/User:Pmlineditor
29 http://en.wikibooks.org/wiki/User:Psoup
30 http://en.wikibooks.org/wiki/User:QuiteUnusual
31 http://en.wikibooks.org/wiki/User:Ramac
32 http://en.wikibooks.org/wiki/User:Recent_Runes
33 http://en.wikibooks.org/wiki/User:Red4tribe
34 http://en.wikibooks.org/wiki/User:Redrocketboy
35 http://en.wikibooks.org/wiki/User:Reece
36 http://en.wikibooks.org/wiki/User:Sigma_7
37 http://en.wikibooks.org/wiki/User:Taxman
38 http://en.wikibooks.org/wiki/User:Thenub314
39 http://en.wikibooks.org/wiki/User:Webaware
40 http://en.wikibooks.org/wiki/User:Xania

List of Figures

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- LGPL: GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[41]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

41 Chapter 38 on page 251

1	EugeneH, Lcarsbot, MonoBot, Webaware	
2		
3	Kwhitefoot, MonoBot, Webaware	
4	Webaware	
5	Webaware	
6	Webaware	
7	Webaware	
8	Az1568, Kwhitefoot, Webaware	

38 Licenses

38.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices". * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy

both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

38.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

38.3 GNU Lesser General Public License

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below. 0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work. 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL. 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

* a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or * b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

* a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

* a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License. * b) Accompany the Combined Work with a copy of the GNU GPL and this license document. * c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document. * d) Do one of the following: o 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source. o 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version. * e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

* a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License. * b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.