



A Siemens Business

---

# **MagNet/ElecNet/ThermNet**

## **Scripting Manual**

April 2018

---

**© 2018 Mentor Graphics Corporation**  
**All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**U.S. GOVERNMENT LICENSE RIGHTS:** The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

**TRADEMARKS:** The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: [mentor.com/trademarks](http://mentor.com/trademarks).

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

**End-User License Agreement:** You can print a copy of the End-User License Agreement from: [mentor.com/eula](http://mentor.com/eula).

Mentor Graphics Corporation  
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.  
Telephone: 503.685.7000  
Toll-Free Telephone: 800.592.2210  
Website: [mentor.com](http://mentor.com)  
Support Center: [support.mentor.com](http://support.mentor.com)

Send Feedback on Documentation: [support.mentor.com/doc\\_feedback\\_form](http://support.mentor.com/doc_feedback_form)

# TABLE OF CONTENTS

<b>Chapter 1: Introduction</b>	<b>5</b>
The Goal of Scripting .....	5
User Scripts.....	5
Session log .....	6
<b>Chapter 2: Introduction to Coding</b>	<b>7</b>
Programming Conventions .....	7
Program Comments .....	7
Wrapping Lines of Code .....	7
Case-Sensitivity .....	7
Infolytica Scripting Commands .....	8
Implementing User Scripts.....	8
First Sample Program .....	8
<b>Chapter 3: Intermediate Coding</b>	<b>11</b>
Subs and Functions .....	11
Some Built-In VBScript Functions .....	12
Call Statement .....	13
Declaring Constants and Variables.....	13
Dim .....	14
Option Explicit.....	14
Declaring Arrays .....	14
ReDim .....	15
UBound and LBound and Multi-dimensional Arrays .....	15
Erase.....	16
Array Command.....	16
Variable Manipulation.....	16
Scope of a Variable.....	18
Reading Inputs from Input Boxes.....	19
Message Boxes and Outputs.....	21
The Chr and Asc Functions .....	23
The If-Then-Else and Select Case Statements .....	23
Control Loops .....	28
Do-Loops .....	29
Nesting Loops.....	29
<b>Chapter 4: Advanced Programming</b>	<b>30</b>
Error Handling .....	30
Syntax Errors .....	30
Semantic Errors.....	30
Run Time Errors .....	31
Exit Statement.....	32
Run Time Error Handler .....	32
ActiveX Software Interaction .....	35

<b>Chapter 5: Tips for Scripting With Infolytica Products</b>	<b>37</b>
Scripting Interfaces .....	37
Scripting Commands.....	39
Passing Arrays as Input Values .....	40
Infolytica Specific Event Handlers.....	40
Important Infolytica APIs for Accessing Third Party Software .....	41
ProcessCommand API .....	41
GetVariant and SetVariant APIs.....	42
RunScript API .....	42
 <b>Appendix A</b>	 <b>43</b>
JScript .....	43
References.....	43
Script Forms.....	43

# Chapter 1: Introduction

## THE GOAL OF SCRIPTING

The goal of the scripting feature, within any of the three Infolytica products (MagNet, ElecNet, ThermNet), is to provide an extensible, industry-standard tool to automate modeling, solving, and post-processing tasks. All three products can also be customized to meet user requirements, as well as enable a direct link between the application and any other ActiveX compliant software, such as Microsoft Word or Microsoft Excel.

Scripting helps to automate the process of building models. Complex and repetitive tasks can be simplified with scripting. In addition to building models, scripting can be used to solve models and to extract post-processing information.

## USER SCRIPTS

Within the various Infolytica applications, scripts can be executed within MagNet, ElecNet, and ThermNet.

Scripts are text files that contain a series of scripting commands. Script files, which can be created or modified in any text editor, can be automatically generated using the Infolytica application and later modified by the user. Alternately, new scripts may be written entirely from scratch in a text editor. Note that when editing or creating a user script in a text editor, one must make sure to save the file in text format with the .vbs extension, in order to have it recognized by the Infolytica application. There are two methods to creating scripts. The first method requires no additional coding from the user. The script can be used to build or solve models within any of the products.

In order to automatically create a user script, select 'Start Recording User Script...' from the Scripting Menu and save the script. The application then records all of the commands and actions carried out by the user until the user stops or pauses the script (**Note:** Any actions carried out by any of the Infolytica Extensions are not recorded.). One can then run the script by selecting 'Run Script...' from the Scripting menu and choosing the desired script file. These recorded user script files can be used as a starting point in generating scripts that are more functional.

## Session log

MagNet, ElecNet and ThermNet automatically generate a script file ('MagNetSessionLog.vbs', 'ThermNetSessionLog.vbs' or 'ElecNetSessionLog.vbs') every time they are accessed. This session log, located, by default, in the LogFiles folder of the application, can also be used as a user script and run from within the application. The file contains a log of all the activities from the moment one enters the program until the program is closed (**Note:** The session log is backed up for up to ten times that the application is opened before the file is removed. The phrase 'Backup-' followed by a number is appended to the front of the session log file name). One can view the contents of the session log file by selecting 'View Session Log' from the help menu.

The second method of implementing script files involves customizing existing scripts. A script file can be modified using a text editor, such as Word or Notepad. The values in the script can later be altered and used in some of the code or whole sections of code can be eliminated altogether. In addition, variables and program coding can be added to increase the functionality of the script. One can add selected portions from multiple scripts together in order to increase the functionality of a script or create an entirely new script.

Scripts can be written in either VBScript or in JScript. The coding language of both the user scripts and the session log for MagNet, ElecNet and ThermNet is in VBScript. Therefore, this manual will focus on VBScript programming syntax.

# Chapter 2: Introduction to Coding

## PROGRAMMING CONVENTIONS

### Program Comments

Comments are lines of text in code, which are ignored by the program once it is run. Comments are single lines preceded by an apostrophe or by the statement 'REM'. The user could implement comments in order to write notes to himself or place apostrophes at the beginning of lines of code, commenting them out when error checking, effectively erasing the lines of code to the program. If comments are not preceded by an apostrophe or a 'REM' statement, the program would try to interpret the text and an error would be indicated when the program is run.

It is a good programming practice to put comments throughout code. Comments could indicate which parts of the code are responsible for which functions such that the user can readily understand the function of the different elements of the code and modify it easily.

### Wrapping Lines of Code

Another important symbol that can be used in programming is the ' \_ ' symbol. This symbol allows a command line to be continued from one line to the next to avoid having long commands. The ' \_ ' symbol must be preceded by a space and does not work within strings. This symbol is useful when calling functions or subs that require many input parameters. The following example shows how the symbol may be used.

```
SubName(Parameter1, Parameter2, Parameter3, _  
Parameter4)
```

### Case-Sensitivity

It is important to note that VBScript language is not case-sensitive. Therefore, if one were to write "Parameter1", "parameter1" or "PARAMETER1" within the code, the program would interpret all of them as referring to the same variable. This trait is true for key words in VBScript (such as Sub and If) as well as user defined terms, such as function and variable names. Capitalization of letters within the code is strictly optional and is done only to make the code more easily readable to the user. However, when comparing two similar text strings with each other, case sensitivity does become an issue. The two strings, "String1" and "string1" for example, would not be considered equal by the program unless the capitalization was identical.

## INFOLYTICA SCRIPTING COMMANDS

Scripts are most useful when they interact with MagNet, ElecNet or ThermNet files. This is done by using the specialized Infolytica scripting commands that are available with the scripting option. For example, there are scripting commands to open new files, draw construction slice edges, create components, and animate models within the Infolytica products. With the scripting commands, one has access to all of the pre-processing, solving, and post-processing capabilities that are present in the applications.

The complete list of scripting commands, applicable to each of the Infolytica products, is available in the scripting reference pages of their respective online help files. Integrating these scripting commands with the VBScript commands, one can write, save, and extract information to or from a MagNet, ElecNet or ThermNet. (**Note:** In order for scripting to interact with a document, the file must be open. Documents can be opened from within the script itself).

Additional tips for scripting with MagNet, ElecNet, and ThermNet can be found in [Chapter 5: Tips for Scripting with Infolytica Products](#).

## IMPLEMENTING USER SCRIPTS

A simple method of coding to interact with MagNet, ElecNet or ThermNet is to record a script and implement it within another script. The session log will automatically record all commands used within the application that, in turn, can be used to automate processes. Alternatively, one can cut & paste sections from the application's session log.

## FIRST SAMPLE PROGRAM

### RECORDING A USER SCRIPT

1. On the Scripting menu, click 'Start Recording User Script...'.  
  
This will prompt for a selection of a name for the script.

2. Navigate to a drive and directory where the script file should be saved. Enter USER SCRIPT and click on 'Save'.

3. On the File menu, click New.  
  
If a MagNet, ElecNet or ThermNet file is already open, the program will prompt for the file to be saved before continuing.

4. On the Tools Menu, click 'Keyboard Input Bar'.

5. On the Draw menu, click 'Line'.



6. In the Keyboard Input Bar enter the following co-ordinates:
  - 0, 0 Press ENTER
  - 0, 10 Press ENTER
  - 10, 10 Press ENTER
  - 10, 0 Press ENTER
  - 0, 0 Press ENTER
7. Press ESC.
8. Select the construction slice surface.
9. On the Model menu, click 'Make Component in a Line'.
10. In the Make Component in a Line dialog, click OK.
11. On the Scripting menu, click 'Stop Recording User Script'.

Once this series of actions is completed, the user script generated can be run on its own. To run the script on its own, simply select 'Run Script' on the Script Toolbar or in the Scripting menu. Once the dialog box opens, navigate to the file "User Script" and click on Open. The script will then repeat the above series of actions and display the image in the view window.

Note that the code in the recorded user script is merely a subset of commands that are written out to the session log of the application. The contents of the various session logs of the application can be viewed by clicking 'View Session Log' in the help menu, where AppName would be MagNet, ElecNet or ThermNet, depending on which application is open. The session log will then appear in a new window, using the default text editor. Portions of this file can be copied and used in other code. Note that it is recommended that this file not be modified while the application is still running.



# Chapter 3: Intermediate Coding

## SUBS AND FUNCTIONS

A sub refers to a subroutine that does not return a value. It executes a series of statements and then it returns control to the point in the program from which it was called. The syntax for a sub is:

```
Sub SubName(parameters)
    Code
End Sub
```

A function, on the other hand, returns a value to the point that it was called in the program. Its structure is similar to that of subs. In order to return the value calculated by the function to the point from which it was called, the name of the function must be equated to the result obtained.

```
Function FunctionName(parameters)
    Code
    FunctionName = Result
End Function
```

Note that the names of both functions and subs cannot contain any spaces. The names must start with a letter and may include underscores and numbers.

To write a function that squares a number, an input value would be required (i.e., the number to be squared). The function would therefore need parameters in order to complete a task. Parameters are the input values that a sub or a function needs in order to carry out a task. The parameters are enclosed within brackets after the name of the sub or function. If there are no parameters, no parentheses are necessary. Therefore, the function named Square would be written as follows:

```
Function Square (Number)
    Square = Number * Number
End Function
```

If a function or a sub has more than one parameter, the different parameters in the list are separated by commas.

Subs and functions can be called from within other subs or functions. In order to call a sub, one would merely enter the name of the sub along with any input parameters that the sub would require. In order to call a function, a value must be equated to the function call such that the program stores the returned value. Again, the parameters in the list are separated by commas.

```
SubName(parameters)      Value = FunctionName(parameters)
```

Writing subs and functions can help to minimize the amount of coding necessary. If there is a series of statements that must be carried out repeatedly or carried out by different functions, these subroutines can be written once and accessed over and over again instead of being written out repeatedly. Furthermore, by using these subroutines, the code can be broken down

into functional sections such that each subroutine is responsible for a different aspect of the whole. This makes the code easier to read for the user; thus, the code can be more easily modified.

## Some Built-In VBScript Functions

Some functions are already written within VBScript that can be used in writing new code. These include:

Function	Description
<b>IsNumeric(Text)</b>	Returns True if the text can be interpreted as a number and returns False otherwise. Inputs of the form 3E5, representing $3 \times 10^5$ , are considered to be numerical.
<b>IsDate(Text)</b>	Verifies that the text input can be interpreted as a date. Some examples from among the several acceptable date formats include – January 3, 1999; Jan. 3, 99; 1/3/1999; 01/03/99; 1-3-1999; 01-03-99.
<b>UCase(Text)</b>	Converts all characters in a text string into upper case.
<b>LCase(Text)</b>	Converts all of the characters in a text string into lower case.
<b>Trim(Text)</b>	This function trims the spaces from the beginning and the end of a text string. Alternatively, one can use RTrim or LTrim in order to remove blank spaces from the right or the left of a text string, respectively.
<b>Time</b>	Function that returns the current time of the system.  <b>CAUTION:</b> Alternatively, this can be used in order to change the system clock of one's computer. One must be careful in using this function.
<b>Date</b>	Function that returns the current date of the system.
<b>Now</b>	Function that returns the current time and date of the system.
<b>Timer</b>	Function that returns the number of seconds that has elapsed since midnight.
<b>Mid(string, start, length)</b>	This function takes a string and returns a string of length 'length' starting at position 'start' in the string. The first letter in the string is considered position 1.
<b>Left(string, length)</b>	This returns the first length number of characters from the left of the string. If length is greater than or equal to the length of the string, the entire string is returned.
<b>Right(string, length)</b>	This function returns the last length number of characters from the right of the string. If length is greater than or equal to the length of the string, the entire string is returned.
<b>InStr(string, substring)</b>	This function looks for the occurrence of a string 'substring' within a larger string. It returns the position within 'string' in which the shorter string is found. If there is no matching string, the function returns zero.

The above commands are among the most commonly used commands used to manipulate data with VBScript. Further info can be found on Microsoft's VBScript reference page at: <http://msdn.microsoft.com/en-us/library/3ca8tfek.aspx>

## Call Statement

The call statement is one way to transfer control from the currently executing section of code, to a new sub or function. Often, when writing scripting code, one sees the call statement being used to call subs or functions. The call statement merely transfers control to the listed subroutine or function. However, it is important to note that when used with functions, the return value is then omitted, effectively turning the function into a subroutine.

**Call** SubName(parameters)

**Call** FunctionName(parameters)

The call statement can be used to invoke both Infolytica scripting APIs and subs and functions written in VBScript. In the session logs, Infolytica scripting APIs are invoked using the call statement. When accessing Infolytica scripting APIs, it is best to use the call statement for all subroutines since they do not return a value, or function calls in instances where the return value is not needed.

## DECLARING CONSTANTS AND VARIABLES

Constants are usually values within formulas that need to be used over and over again. For example, if a program needs to convert from radians to degrees repeatedly, one would need to use the value of pi often. Instead of writing out the value 3.14159... each time it is required, one can merely define a constant and use it instead. To declare a constant, the following syntax would be used:

**Const** Name = value

Therefore, in order to declare the value of pi, one would write “Const Pi = 3.14159”. Once a constant is declared, anytime that the value of pi was required within a program, one would substitute the name applied to the constant, in this instance ‘Pi’. Note that since there is no case sensitivity in VBScript, one could also use the declaration ‘pi’ in order to access the same value. The value of a constant remains unchanged throughout the run of the program.

Variables are items used to store data within a program. For example, one could simply declare a variable and equate it to the value being read from VBscript’s InputBox function. Unlike constants, the values of variables can be changed repeatedly within code. This makes writing programs to manipulate user inputs much less cumbersome.

## Dim

In order to declare a variable, one uses the Dim command. The Dim command reserves space in memory such that the variable can store any type of data.

### **Dim Variable**

The variable here is automatically declared as a variant such that it can store a string, numerical data, an array or any other variable type. Therefore, in order to store information, the following statements would be used:

### **Dim XValue**

```
XValue = 10
```

The variable XValue would then contain a value of 10.

## Option Explicit

In VBScript, one may use variables without first defining them. Therefore, in the statement above, one may use the command 'XValue = 10' without first using the Dim statement and obtain the same results. In order to have a program that only allows the use of variables that have been previously declared, the 'Option Explicit' command must be used.

The 'Option Explicit' command is placed before all other statements in a user script, apart from any sub or function. The following series of statements

### **Option Explicit**

### **Sub Undeclared**

```
XValue = 10
```

### **End Sub**

will fail, generating an error message when the program is run, since the variable 'XValue' is not defined prior to its being used. The variable must first be defined using the Dim statement, in order to be recognized by the program.

The advantage to using the Option Explicit command is that it prevents one from misspelling the names of variables. In the following program excerpt, if the Option Explicit command is not used, then the program would execute, but the variable XValue would not contain any useful information.

### **Dim ValueX**

```
ValueX = 10
```

```
Evaluate (XValue)
```

As written, the variable XValue is passed on to the sub 'Evaluate', although it has not been initialized and contains no useful data. It is accidentally passed on to the sub instead of the variable ValueX. This may cause the sub 'Evaluate' to fail since it does not have the expected input. With the Option Explicit command, the program alerts the user to the use of the incorrect variable name.

## Declaring Arrays

An array is a special type of variable in that it can hold several different values at once. In order to declare an array that can hold a given number of values, one need only enter the index of the last element in the array in parentheses directly after the variable declaration. Since the first element in the array has an index value of zero, one would declare a value one

less than the total number of elements in the array. Therefore, to declare an array with 10 values, the following statement would be used:

```
Dim ArrayName(9)
```

The indices of the data within the array would range from zero to nine inclusive, a total of 10 separate values.

## ReDim

Once an array is declared with a given number of inputs, the same array name can later be declared again with a different number of indices using the ReDim command. Furthermore, the ReDim command can be used to declare a previous variant variable as an array.

A) <b>Dim</b> ArrayA(4)	B) <b>Dim</b> ArrayB	C) <b>Dim</b> ArrayC()
<b>ReDim</b> ArrayA(9)	<b>ReDim</b> ArrayB(9)	<b>ReDim</b> ArrayC (9)

Figure 1 *The different methods for declaring an array with the ReDim statement*

- a) Resizing an existing array; b) Converting from a variant to an array;
- c) Adding the dimension of an empty array

When inputting values into an array, one need only declare the name of the array and the index and equate it to a value. The statement

```
ArrayName(0) = 4
```

places the number 4 in the first position of the array. Each index of a given array can contain values of different types such as strings, integers, dates, etc.

When resizing an array with the ReDim command, all of the information contained in an array is lost. In order to preserve the contents of an array while it is being resized, the 'Preserve' command would need to be used.

```
ReDim Preserve ArrayName(15)
```

This function call resizes an already existing array to another size while keeping all of the information that was previously stored in the array. Note that this statement can only be used to increase the size of an array and not to decrease it.

## UBound and LBound and Multi-dimensional Arrays

Some of the other functions associated with arrays are UBound and LBound that return the upper and lower bound of the array respectively. Since LBound always returns 0, the lower bound of the array, UBound returns one less than the number of elements in the array. The syntax of the UBound and LBound commands is

```
Value = UBound(ArrayName)
```

The last index of the array is then stored in the variable 'Value'.

Multidimensional arrays may be declared in order to represent matrices. In order to declare arrays with multiple dimensions, the number of entries in each dimension would need to be separated by a comma.

```
Dim ArrayName(9,4)
```

The above is a declaration of a 10 x 5 matrix. The array has two dimensions, each dimension containing five values. Note that using the UBound statement on a multi-dimensional array by default returns the upper bound of the first dimension. Therefore, in the above statement, UBound would return '9'.

In order to use UBound and LBound to return the bounds of other columns, a second, optional parameter must be passed to the function, indicating which column one desires the bounds for

```
Dim ArrayName(10, 20, 30)
Column1 = UBound(ArrayName, 1)
Column2 = UBound(ArrayName, 2)
Column3 = UBound(ArrayName, 3)
```

In the above example, the values of Column1, Column2 and Column3 are 10, 20 and 30 respectively. Note that the columns are numbered starting from 1 and not from 0.

## Erase

In order to delete the contents of an array, one would use the Erase command. The syntax for the Erase command is

```
Erase ArrayName
```

The dimensions of the array remain unchanged after the erase command is executed.

## Array Command

One must take care not to name arrays 'Array'. This is because Array is a function that is defined within VBScript that generates an array from a given list of elements. The statement

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

creates an array called Count, which contains six elements – 0, 1, 2, 3, 4 and 5. These elements are placed in the array, in the listed order. Note that the array can contain values of different types such as strings, integers and dates.

# VARIABLE MANIPULATION

When a program contains a variable, this variable can be manipulated by the program code. In addition to equating different values to a variable, in the case of strings, different strings can be concatenated to form one long string. In the case of variable containing numerical values, one can perform mathematical operations.

In order to concatenate two or more strings, merely use the '&' or the '+' symbol between the individual strings in order to add them together.

```
String1 = "Matter cannot be created"   String2 = " or destroyed."
String3 = String1 & String2
```

The effect of this is that 'String3' contains the value "Matter cannot be created or destroyed." Alternatively, one could have written

```
String2 = String1 & " or destroyed."
```

In this instance, the long string is stored in the variable String2. Note that the string " or destroyed." starts with a space instead of with text. This is because when strings are concatenated, spaces are not placed between the strings. Therefore, when coding, the user has to account for this and enter spaces between concatenated strings wherever necessary.

Strings can be concatenated with other strings as well as to any other type of variable such as integers and dates. All concatenated information will be stored as a string by the program. Note that when variables are concatenated to strings, they are not placed in quotation marks.



For numerical inputs, the variables can be manipulated using arithmetic operations. In addition to the basic mathematical operations, VBScript also contains a range of mathematical functions which one can use to calculate values.

Symbol	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Integer Division (removes all values beyond the decimal point)
<b>Mod</b>	Returns the modulus (remainder) of a division
^	Exponent (e.g. $2^3 = 2^3$ )
<b>Cos(value)</b>	Cosine – Input value is in radians
<b>Sin(value)</b>	Sine – Input value is in radians
<b>Tan(value)</b>	Tangent – Input value is in radians
<b>Atn(value)</b>	Arctangent – Input value is in radians
<b>Log(value)</b>	Natural Logarithm
<b>Exp(value)</b>	Exponential function ( $e^{\text{value}}$ )
<b>Sqr(value)</b>	Square Root
<b>Abs(value)</b>	Absolute Value

Other functions that may be used to manipulate variables include:

Symbol	Function
<b>StrReverse(String)</b>	This function takes a string and returns its reverse.
<b>Len(String)</b>	Returns the length of a string.
<b>DateAdd(interval, amount, date)</b>	This function returns the date with a given interval of time added. The interval is a string indicating the unit of time to add:  “yyyy” adds a year, “q” adds a quarter, “m” for month, “ww” for week, “d” for day, “h” for hour, “n” for minute, and “s” for second. The second input parameter is the amount of time one wishes to add. The last input parameter indicates to which date or time the given amount it added.
<b>DateDiff(interval, date1, date2)</b>	This function determines the amount of time elapsed between two dates, measured in units of time specified by the interval input parameter. The interval values are the same as those for the DateAdd function.

## SCOPE OF A VARIABLE

If a single variable or a constant needs to be accessed by more than one subroutine, then the variable can be declared outside of all subroutines. Variables that are defined outside all subs and functions are said to have a scope that includes the entire program and are referred to as global variables. The scope of a variable indicates the region in which the variable can be accessed. Variables and constants that are defined within one sub cannot be accessed in another sub unless it is passed as an input parameter.

<b>Sub</b> Subroutine1 <b>Dim</b> Number Number = 4 <b>End Sub</b>	<b>Sub</b> Subroutine2 <b>Dim</b> Value Value = Number + 4 <b>End Sub</b>
---	--

If one were to try running the above code and call on Subroutine2, an error would result because Subroutine2 does not have access to a variable ‘Number’ and as such, would be unable to initialize the variable ‘Value’. In order to be accessible to both subroutines, the variable would have to be declared globally, or the value of Number would have to be passed as a parameter from one sub to the other.

Another issue with the scope of the variable is with passing parameters from one subroutine to another. All modifications done to a parameter within a subroutine are local unless the routine is a function and passes the modifications along to the point in the program from which it was called.

<b>Sub</b> Subroutine1 <b>Dim</b> Number Number = 4 Subroutine2 (Number) MsgBox (Number * Number) <b>End Sub</b>	<b>Sub</b> Subroutine2 (Number) Number = Number + 4 <b>End Sub</b>
---	--

In the above code, when Subroutine1 is called, the output of MsgBox will be ‘16’ and not ‘64’. This is because all input parameters to a subroutine, be it function or sub, are merely copies of the original. Therefore, Subroutine2 only makes changes to its own copy of the variable ‘Number’ while the original remains unchanged. In order to get the value ‘64’ the following changes need to be made.

<b>Dim</b> Number	<b>Sub</b> Subroutine1
	Number = 4
<b>Sub</b> Subroutine2	Subroutine2
Number = Number + 4	MsgBox = (Number * Number)
<b>End Sub</b>	<b>End Sub</b>

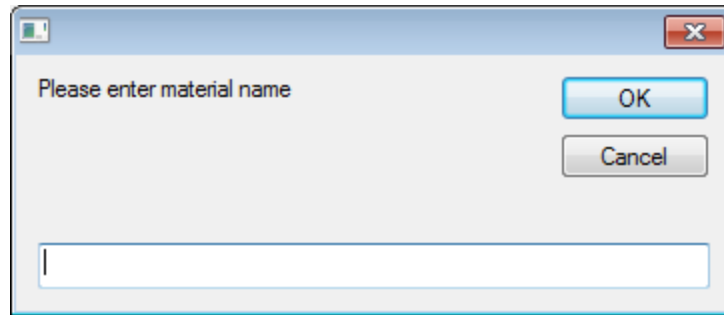
This way, when Subroutine1 is called, the value in the message box reads '64' (**Note:** The statement 'Number = 4' can be located either outside or within the subroutine to obtain this result). If Subroutine2 were to keep its parameter list, then the subroutine would still only contain a local copy of the variable and the message box would display the value '16'.

## READING INPUTS FROM INPUT BOXES

Data is obtained from a user through input boxes. A user may interact with a script by placing information in an input box that can then be read in by the program and stored as a variable. However, all data within an input box is automatically considered as a string, a series of characters. Therefore, if a number were entered within an input box, the program would treat this input no differently than if a series of letters was entered. To have the program treat the input as a number or another type of variable, use the conversion functions included with VBScript to format the input.

Command	Function
<b>CBool(VariableName)</b>	This function converts a value into a Boolean variable (either True or False. This works with numerical values and with the strings "True" and "False").
<b>CDate(VariableName)</b>	This function converts the input value into a date.
<b>CDbl(VariableName)</b>	This function returns the input value and regards it as a double-precision floating-point number.
<b>CInt(VariableName)</b>	This function returns the input value and treats it as an integer. If the number entered is a float, this function would round to the nearest integer.
<b>CLng(VariableName)</b>	This function converts the input into a long variable. If the number entered is a float, this function would round to the nearest long value.
<b>CSng(VariableName)</b>	This function returns the input value and regards it as a single-precision floating-point number.
<b>CStr(VariableName)</b>	This function converts the input into a string. Inputs in text boxes and input boxes are interpreted as strings by default.

These pre-defined functions can convert a variable from one type to another. However, if a series of letters were input into a text box and then the CInt or CDbl command were to be tried, the program would be incapable of interpreting this information. The program would register an error and would cease to execute the code, notifying the user of an error in the code that needs to be rectified.



Input box

An input box is one of the features of VBScript and it can only take in one input at a time from a user. An input box is a function that outputs a screen to the user and returns the user's input. In order to request input from a user using an input box, the following syntax would be used:

```
VariableName = InputBox ("Please enter the material name")
```

The value input by the user is held in 'VariableName' and can then be manipulated by the code once the user hits 'OK'. If the user hits on the Cancel button, then an empty string, "", is returned.

Alternatively, one does not have to equate a variable name to the input. As with other functions, the input box could be manipulated as though it were a value itself. However, if a function is used in this manner, the return value can only be used once, since it is not stored and cannot be accessed again. For example, the following code verifies an input string:

```
If (InputBox("Please enter your name. ") = "Bob") Then
    MsgBox ("Good Afternoon, Bob ")
Else
    MsgBox ("You are not authorized to use this system.")
End If
```

Note that when comparing two strings to each other, the capitalization of letters is taken into account.

In order to change the title of the input box from the default ('Visual Basic'), a second string needs to be included in the declaration of the input box. If one wishes to declare the input box with default text written within as input, then a third input parameter must be added to the declaration.

```
VariableName = InputBox ("String Message ", "Input Box title ", "Default input")
```

In the above declaration, unless the input in the box is changed by the user, the value of 'VariableName' will be "Default input". Both the title and the default input are optional parameters. If the input box was declared without any parentheses, then it is interpreted as a sub instead of a function and it does not return a value, producing the same effect as using the call statement.

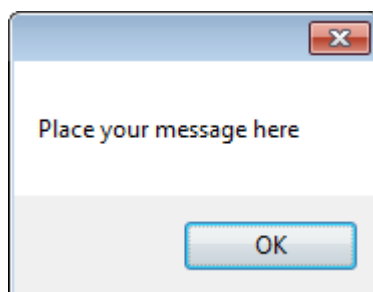
```
InputBox "String Message"      same as      Call InputBox("String Message")
```

## MESSAGE BOXES AND OUTPUTS

Message boxes are the output equivalent to input boxes in VBScript. Like input boxes, they are functions. Message boxes are usually equated to a variable, but they need not be. Message boxes are small windows in VBScript that output messages to the user. By default, message boxes have an 'OK' button. The types of buttons that appear in a message box can be changed. Each different type of button has its own value. This value is returned from the Message Box function, indicating to the program code which button was clicked by the user. Since the default form for a message box only has one button, in this instance, it is not necessary to retain the return value and a message box can be called using the following notation

```
MsgBox("Place your message here")
```

When this is called, a message box will appear with the string appearing as the message and the OK button appearing by default.



**Message Box**

When a message box appears, it suspends all activity of an application. Therefore, other elements of a given program cannot be accessed until a message box is closed by clicking on one of its buttons.

Apart from the default type of message box, in all other types of message boxes there is more than one button that the user may click. In this instance, it is important to note which button the user presses and have the program respond accordingly.

In order to call these different types of message boxes, the following function call could be used:

```
RtnVal = MsgBox("Message String", MsgBoxType, "Message Box Title")
```

The second field in the declaration determines the type of buttons that appear on the message box. The last field in the statement is optional and it changes the title of the message box (the default title is 'Visual Basic'). The different types of message boxes are outlined in the table below.





Message Box Type	Buttons	Return Values
<b>VbAbortRetryIgnore</b>	Contains the 3 buttons 'Abort', 'Retry' and 'Ignore'	VbAbort VbRetry VbIgnore
<b>VbOkCancel</b>	This contains both 'OK' and 'Cancel' buttons	VbOK VbCancel

Message Box Type	Buttons	Return Values
<b>VbOkOnly</b>	This only contains an 'OK' button. This is the default message box type	VbOK
<b>VbRetryCancel</b>	This displays the 'Retry' and 'Cancel' buttons	VbRetry VbCancel
<b>VbYesNo</b>	Displays 'Yes', and 'No' buttons	VbYes VbNo
<b>VbYesNoCancel</b>	Displays 'Yes', 'No' and 'Cancel' buttons	VbYes VbNo VbCancel

If the message box is declared without parentheses about the input parameters, it would be interpreted as a sub and it would not return a value.

In addition, with the different types of buttons, different symbols can be added into the boxes, such as exclamation marks and question marks. In order to add these symbols in a message box, the symbol must be declared after the button type of the message box. Only one symbol can be declared per message box.

**RtnVal = MsgBox("Message String", MsgBoxType & SmbIType, "Message Box Title")**

Symbol	Symbol Type
	VbQuestion
	VbCritical
	VbExclamation
	VbInformation

**Message Box Symbols**

## The Chr and Asc Functions

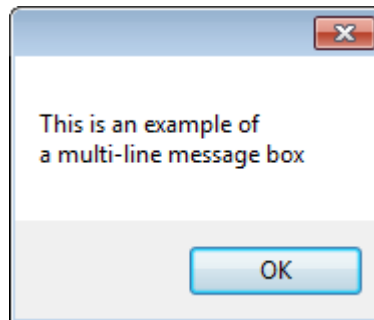
By default, messages boxes output the message string in a single line. In order to have a multiple-lined message, in either a message box or a text box, the 'Chr' function needs to be used. The 'Chr' function returns a single ASCII character based on the input value that is passed in. These output characters can then be output in a text message. In addition to the printable characters located on the keyboard, the ASCII characters provide for non-printable characters such as tabs and line carriages. There are 256 elements returned by the function, based on inputs ranging from index 0 to 255. The non-printable characters most commonly used are

- Chr(9) – Horizontal Tab
- Chr(10) – Carriage Return (New Line)
- Chr(11) – Vertical Tab
- Chr(13) – Line Feed
- Chr(34) – Quotation marks

In order to append a character returned from the Chr function, one needs only use string concatenation. For example, the code

```
MsgBox ("This is an example of " & Chr(10) & "a multi-line message box")
```

gives rise to the following message box



The opposite of the Chr function is the Asc function. This array takes as an input argument a single character and returns the numerical ASCII code for the given input. If an entire string is given as an argument to the Asc function, only the ASCII code for the first character is returned for the string. Therefore the two following function calls

```
ReturnValue = Asc("I")
```

```
ReturnValue = Asc("Input string")
```

would return the same value, 73, the ASCII code for the letter 'I'.

## THE IF-THEN-ELSE AND SELECT CASE STATEMENTS

When coding, at times there are lines of code that should only be executed when a certain criterion is met. For example, one may wish to process data only if the magnetic flux density at a given point is larger than a given value. The If-Then-Else statement is a structure that will execute a series of statements contained within its structure, only if the criterion set within the control statement is met.

The control statement is a statement that can evaluate to either True or False. The conditional code will be executed only if the control statement evaluates to True, otherwise, the code specified in the else section of the structure will be executed. Within control statements, the most commonly used symbols are mathematical expressions of equality and inequality.

Symbol	Meaning
<	Less than
>	Greater than
<=	Less than and equal to
>=	Greater than and equal to
<>	Not equal to
=	Equal to

Mathematical Symbols

The equality symbol can be used to compare two strings of text as well as two numerical values.

In instances where more than one criterion must be met, two or more statements can be compounded using the compound operators found in the table below:

Compound Operators	Property
<b>And</b>	In a control statement, it is True if both statements are True
<b>Or</b>	In a control statement, it is True if either of the two statements preceding is True.
<b>Xor</b>	Exclusive Or. In a control statement, it is True only if exactly one of the two statements is True.
<b>Not</b>	This is True only when the statement that it precedes is False. It toggles the value of a Boolean statement.

Together, these mathematical and compound operators can be used to form control statements for the If-Then-Else statement. The structure of the statement is

```
If (Data >0 And Data <> 10) Then
    Data = Data -1
Else
    Data = Data + 2
End If
```

This structure allows for two alternative series of actions. Only one of these alternates can be executed at a time. If it is found that the control statement **Data > 0 and Data <> 10** is True, then the statements following the 'If' clause are executed. If the control statement is False, then the statements following the 'Else' clause is executed. Therefore, only one of the sets of statements is executed when the script is run, dependent upon the Boolean value of the control statement.



This structure is additionally flexible in that the ‘Else’ clause of the statement is optional. In these instances, there is no alternative code to be executed. In these cases, the syntax of the statement is

```
If (Control Statement) Then  
    Code  
End If
```

In addition to the mathematical symbols used in control statements, calls to functions that return Boolean (True or False) values can also be used in control statements.

<pre><b>Sub</b> UseFunction()     <b>If</b> (ReturnBoolean) <b>Then</b>         Code     <b>End If</b> <b>End Sub</b></pre>	<pre><b>Function</b> ReturnBoolean()     ReturnBoolean = True <b>End Function</b></pre>
---	---

If there are more than two possible outcomes, one can nest the If-Then-Else statements within each other. However, this can become rather bulky since each ‘If’ statement needs a corresponding ‘End If’ statement. Instead of using several ‘Else’ clauses followed by separate ‘If’ clauses, the term ‘ElseIf’ can be used instead. This eliminates the need to use several ‘End If’ statements that can be hard to keep track of.

<b>Nesting If-Then-Else Statements</b>	<b>If</b> (Operator = "+") <b>Then</b> Answer = Op1 + Op2 <b>Else</b> <b>If</b> (Operator = "-") <b>Then</b> Answer = Op1 - Op2 <b>Else</b> <b>If</b> (Operator = "*" Or Operator = "x") <b>Then</b> Answer = Op1 * Op2 <b>Else</b> <b>If</b> (Operator = "/") <b>Then</b> Answer = Op1 /Op2 <b>End If</b> <b>End If</b> <b>End If</b> <b>End If</b>
<b>Nesting If-Then-Else Statements Using ElseIf</b>	<b>If</b> (Operator = "+") <b>Then</b> Answer = Op1 + Op2 <b>ElseIf</b> (Operator = "-") <b>Then</b> Answer = Op1 - Op2 <b>ElseIf</b> (Operator = "*" Or Operator = "x") <b>Then</b> Answer = Op1 * Op2 <b>ElseIf</b> (Operator = "/") <b>Then</b> Answer = Op1 /Op2 <b>End If</b>

The 'ElseIf' clause

An alternative to nesting several If-Then-Else statements together is to use the select case statement. The select case statement is a special type of statement that allows for multiple outcomes. In the If-Then-Else structure, control statements evaluate to True or False allowing a maximum of two different possibilities, one set of actions if the control statement evaluates to True and another if it evaluates to False. However, a select case statement can have several different alternatives.

The syntax for the Select Case statement is as follows:

```

Select Case variable
    Case value1
        Statements1
    Case value2
        Statements2
    Case value3
        Statements3
End Select

```

This statement will execute one of three different sections of code depending on whether 'variable' is equal to value1, value2 or value3. If it is not equal to any of these three values, the code is ignored and the statement after the select case statement is executed. Alternatively, a 'Case Else' statement may be added as the last clause. The 'Case Else' clause

is executed when the variable is not equal to of the cases listed in the statement. The select case statement eliminates the need to use nested If-Then-Else statements as demonstrated by the following code (the 'Case Else' clause of the statement is optional).

Nesting If-Then-Else Statement	Select Case Statement
<pre> <b>If</b> (Operator = "+") <b>Then</b>     Answer = Op1 + Op2 <b>ElseIf</b> (Operator = "-" ) <b>Then</b>     Answer = Op1 – Op2 <b>ElseIf</b> (Operator = "*" Or Operator = "x") <b>Then</b>     Answer = Op1 * Op2 <b>ElseIf</b> (Operator = "/" ) <b>Then</b>     Answer = Op1 /Op2 <b>Else</b>     MsgBox("Invalid Operator") <b>End If</b> </pre>	<pre> <b>Select Case</b> Operator     <b>Case</b> "+"         Answer = Op1 + Op2     <b>Case</b> "-"         Answer = Op1 – Op2     <b>Case</b> "*", "x"         Answer = Op1 * Op2     <b>Case</b> "/"         Answer = Op1 / Op2     <b>Case Else</b>         MsgBox("Invalid Operator") <b>End Select</b> </pre>

**'If' and 'Select Case' statements**

Note that in the select case statement, there may be multiple inputs that trigger the same event. However, in cases where a range of values is used as a point of comparison (e.g.  $X > 17$  And  $X \leq 24$ ), one must use the If-Then-Else statement instead.

## CONTROL LOOPS

Control loops are similar to If-Then-Else statements in that they contain a control statement and they are only executed when certain criteria are met. However, unlike the previous structures, control loops will repeat the series of statements until the criterion is no longer met. There are many different types of control loops as outlined in the table below:

Loop Structures	Syntax	Properties
<b>For-Next</b>	<b>For</b> counter = 1 To 20 Value = Value + 1 <b>Next</b>	The body of the loop executes the number of times indicated in the first line of the for statement or until an exit statement is reached.  The 'Exit For' statement may be included within the loop.
<b>For Each-Next</b>	<b>For Each</b> Value in ArrayName MsgBox (Value) <b>Next</b>	This body of the loop executes once for each value in an array or until an exit statement is reached.  Within the loop, each array element, in turn, is assigned to the variable 'Value'.  The 'Exit For' statement may be included within the loop.
<b>Do-Loop</b>	<b>Do</b> Value = Value + 1 <b>If</b> (Value > 20) <b>Then</b> <b>Exit Do</b> <b>End If</b> <b>Loop</b>	In this loop, the exit statement must be included within the statements in the loop or else the loop would continue infinitely. This is accomplished in the example by using an if statement to test for the exit condition.
<b>Do-Loop Until</b>	<b>Do</b> Value = Value - 1 <b>Loop Until</b> (Value < 20)	The body of this loop executes at least once until the control statement at the end of the loop evaluates to True.  The 'Exit Do' statement may be used within the Do-Until loop.
<b>Do- Loop While</b>	<b>Do While</b> (Value < 20) Value = Value + 1 <b>Loop</b>	This loop executes until the control statement is evaluated as False.  The 'Exit Do' statement may be used within the Do-While loop.
<b>While-Wend</b>	<b>While</b> (Value < 20) Value = Value + 1 <b>Wend</b>	This loop executes until the control statement is evaluated as False.

Control loops

## Do-Loops

The Do-Loops are very flexible control statements, not only because of the different variations, but also because the Do-While Loop and the Do-Until loop can be set up such that the control statement is evaluated either at the end or at the beginning of the loop. The difference between these two different arrangements is that once the control statement is placed at the beginning of the loop, the loop verifies the control statement first before proceeding. However, once the control statement is placed at the end, the control loop will execute at least once before evaluating the control statement.

## Nesting Loops

Further control can be added to a program by nesting loops. Nested loops occur when one loop is found within another. This technique can be carried out with loops of similar or dissimilar types.

An interesting point about VBScript is that it can use integers as True and False values. All integers other than 0 are considered to be True and 0 is considered to be False. Therefore, integers can then be used in control statements for the various loops. If one were to input a number called 'Divisor' in an input box, code could be written to verify that 'Divisor' is not zero.

```
Divisor = CInt (InputBox ("Enter a divisor for 12."))  
If (Divisor) Then  
    MsgBox (12 /Divisor)  
Else  
    MsgBox ("You cannot divide by zero.")  
End If
```

This code first reads the input text as an integer and then it verifies that the integer is not zero. If it is zero, then it cannot be used as a divisor. Alternatively, the control statement could have been '(Divisor <> 0)'.

# Chapter 4: Advanced Programming

## ERROR HANDLING

There are three different types of coding errors that one can make: syntactic errors, semantic errors and run time errors. Syntactic errors are errors that use incorrect syntax in the code, such as forgetting to write 'Then' in an If-Then-Else loop or misspelling a function name. Semantic errors are those that arise from incorrect logic. Often, the program will still run but give erroneous output. These often arise in arrays and lists when one forgets that the index of the first item is 0 instead of 1. The third type of error is the run time error. These are errors that arise when the program is run and are typically the result of the program being unable to handle incorrect inputs and other unexpected situations that the programmer didn't account for when coding.

### Syntax Errors

Syntax errors are picked up by the program as soon as one tries to run it. Often, an error can remain undetected if it is located in a line of code that only need to be accessed rarely such as an else clause within an If-Then-Else statement in a sub that is rarely called. Therefore, the program can run several times without registering an error. However, once the program comes upon this line of code and tries to interpret it, the program would cease to run and the program would indicate that there is an error in the code. Syntax errors are easily fixed by checking the syntax of the statements in the code at the location indicated by the error message. Misspelled variables can be detected by using the 'Option Explicit' command in the code. For more information concerning this command, please refer to page 14.

### Semantic Errors

Semantic errors are harder to notice and they are usually discovered only through vigilant observation. If the erroneous output is never noticed, the program would run without generating any error messages. If one notices a semantic error, one can determine where in the code the errors are generated by placing message boxes throughout the code and using them to output some of the values and parameters that the program is working with. Therefore, one could look at the input, output in a systematic manner, determine where in the code the logical flaw is located, and then correct it.

Examples of semantic errors include trying to access an element from outside the range of an array or improperly accessing the fields from within an array.

<b>Dim</b> ArrayName(9)	<b>Dim</b> ArrayName(9)
<b>Dim</b> Count	<b>Dim</b> Count
<b>For</b> Count = 1 To 10	<b>For</b> Count = 1 To 9
ArrayName(Count) = Count	ArrayName(Count) = Count
<b>Next</b>	<b>Next</b>

In both of the above examples, the error is in the assumption that the first element in the array is 1. The code to the left will result in an error message because the largest index of the array is 9, even though it contains 10 elements. Similarly, the code to the right assumes that the number used in the declaration indicates the number of elements that are to be found in the array instead of indicating the index of the last element in the array.

The first semantic error will cause the program to cease running as soon as the variable Count reaches the value 10. The second error, however, may never generate an error if the array is always accessed in the same manner. However, as soon as one accesses and tries to manipulate the value in ArrayName(0), which is not initialized, then the script may stop and generate an error message, depending on the manipulation.

Often, semantic errors arise in control statements for loops and If statements, as shown below

```
If (X < 0 And X > 100) Then
    Code
End If
```

In the above code, the code in the loop will never execute because a value can never be less than 0 and greater than 100 at the same time. Or needs to be used in the control statement rather than And.

## Run Time Errors

Run time errors are those which are encountered when the program is run, usually as the result of input values or a particular order of events triggered by the user. These errors are picked up by the program when incompatible input is entered while the script is run, such as trying to multiply two strings together, thereby causing the program to cease running. An example of a run time error can be demonstrated with a script that carries out division.

With the following code, the script would run correctly for the most part. However, there would be instances when the program would not know how to deal with the given input and the program would crash.

```
Call Division
Sub Division
    Dim Op1, Op2
    Op1 = Cdbl (InputBox ("Enter operand 1"))
    Op2 = Cdbl (InputBox ("Enter operand 2"))
    MsgBox (Op1/Op2)
End Sub
```

The above code would work perfectly only if every time that the script was executed, the expected inputs – two numerical values – were in each Input box. However, several problems that this code does not deal with could cause the program to crash.

- **Op2 = 0**  
A number cannot be divided by zero. Therefore, if zero was entered into the second text box, the program would not be able to carry out the division with the given input. The script would automatically cease to run and an error message would be generated.
- **Nothing is input in the input boxes**  
An error message would be generated indicating invalid input.
- **Text is input in the input boxes**  
The script would not know how to handle the text and when it tries to convert the text into an integer, the program would crash, again generating an error message indicating

invalid input.

These are just some of the problems that one may run into when running a script that requires user inputs. In order to make the program resistant to these possibilities, all of the likely errors and misuses must be taken into account and code written that deals with all of them. Therefore, one method of correcting the code would be as follows:

**Call Division**

**Sub Division**

**Dim Op1, Op2**

*'The following statement verifies that the inputs can be interpreted as numbers. This handles problems 2 and 3.*

Op1 = **InputBox** ("Enter operand 1")

Op2 = **InputBox** ("Enter operand 2")

**If (IsNumeric (Op1) And IsNumeric (Op2)) Then**

Op1 = **Cdbl** (Op1)

Op2 = **Cdbl** (Op2)

**Else**

**MsgBox** ("Please enter numbers in the operand fields")

**Exit Sub**

**End If**

*'The following statement handles problem 1. If the value is 0, the 'If'*

*'statement is ignored and the program goes on to the 'Else' clause of the statement.*

**If (Op2) Then**

**MsgBox** (Op1/Op2)

**Else**

**MsgBox** ("You cannot divide by zero")

**End If**

**End Sub**

## Exit Statement

The Exit statement is used to exit subs, functions and loops before they would ordinarily terminate. Above, the 'Exit Sub' command is used to exit the sub once an error is encountered. The exit statement is particularly helpful with error handling because once an erroneous input or value is obtained, typically the program cannot continue to run and must be exited in order to prevent the program from crashing. The 'Exit Function' statement is used to exit functions.

## Run Time Error Handler

By default, run time errors are fatal, meaning that they will immediately terminate the running code and display an error message. In order to continue running the script beyond the point of error, the user may have to modify the code before running the script again. Even if the code is not modified, the script must be run from the beginning. However, VBScript provides a basic error handler that skips over all lines of code that would ordinarily generate a run time error and continue running the program.

The command line to skip over run time errors is "On Error Resume Next". This command tells the program that when a run time error results from trying to execute a line of code, to skip over the line and continue on to the next line of code. This can be useful to temporarily skip over the program error and determine whether or not the code that appears after the error



is correct or not. The “On Error Resume Next” command turns on the error handling which can be subsequently turned off with the “On Error GoTo 0” command.

It is important to note that the Error statement must be invoked in every sub or function in which one wishes an error to be skipped. If another subroutine is called, then the status of the error handler is not carried over to the called subroutine. In the following example, a run time error is generated when the function DivideByZero is called.

<b>Sub</b> SkipError	<b>Function</b> DivideByZero(input)
<b>On Error Resume Next</b>	DivideByZero = input/0
<b>Dim</b> Three, Answer	<b>End Function</b>
Three = 3	
Answer = DivideByZero(Three)	
<b>On Error GoTo 0</b>	
<b>End Sub</b>	

Since the Error statement is not invoked in the DivideByZero function, when the sub SkipError is run, a run time error message would still be generated and the program will be halted.

The run time error handler can be used in order to debug code. If one wants to verify the output of the code and keeps running into an error, instead of fixing it immediately, problematic code can simply be skipped in order to continue with the program and verify the overall structure.

When the 'On Error Resume Next' statement is used, after every error, the Err object is created. Two properties of the Err object that help determine the type of error are Err.Number and Err.Description. Err.Number is the number by which the error is defined. Err.Description gives a short description of the error. These values are reinitialized every time that an error occurs. In the event that there is no error, Err.Number is 0 and Err.Description contains an empty string. Using the run time error handler, the division subroutine on page 31 can again be rewritten as follows:

```

Call Division
Sub Division
    On Error Resume Next
    Dim Op1, Op2
    Op1 = InputBox ("Enter operand 1")
    ' Verify if the first operand generates a type mismatch error
    If (Err.Number = 13) Then
        MsgBox ("Please enter a number in the operand fields")
        Exit Sub
    End If
    Op2 = InputBox ("Enter operand 2")
    ' Verify if the second operand generates a type mismatch error
    If (Err.Number = 13) Then
        MsgBox ("Please enter a number in the operand fields")
        Exit Sub
    End If
    MsgBox (Op1/Op2)
    ' Verify that there is no division by zero
    If (Err.Number = 11) Then
        MsgBox ("You cannot divide by zero")
    End If
    On Error GoTo 0
End Sub

```

In the above code, the Err.Number property is used to verify the user input. If one were to use the Err.Description property for each error number, 11 would generate a description "Division by zero" and 13 would generate "Type mismatch".

For a complete list of VBScript run time error numbers visit the webpage:

<http://msdn.microsoft.com/en-us/library/xe43cc8d.aspx>

Since the output of Err.Number is zero when there is no error, instead of including the statement 'If (Err.Number = 11) Then ...', one could simply use the code, 'If (Err.Number) Then ...'. The problem with this is that there could be many other errors present in the code that can result in the genesis of an error number. In this instance, if the error number is not specified in the If statement, the message box 'You cannot divide by zero' will always be generated, no matter what values are input by the user.

## ACTIVEX SOFTWARE INTERACTION

MagNet, ElecNet and ThermNet are ActiveX compliant software and as such, one can use scripting in order to interact with other applications that support ActiveX. For example, one can access programs such as Microsoft Word and Excel, from within a MagNet script. Conversely, a MagNet file can be accessed and manipulated from within other ActiveX software, such as MATLAB®. This can be useful in terms of extracting data from a MagNet file to make extensive calculations or draw a graph.

Similar to how Infolytica scripting APIs allows one to execute many different commands in the user interface, there are scripting commands that are affiliated with other programs that can, in turn, be used to access other programs. For example, in order to open the Microsoft Word program from a script, the following commands would be used:

```
Set Word = CreateObject("Word.Application")
Word.Visible = True
Word.Documents.Add
```

The first command line sets the value of the variable 'Word' to a word application. The variable 'Word' is then said to be a handle to the application. The second command makes the word application visible to the user. The last line opens a new, blank document. In order to open an existing document, one would use the command

```
Word.Documents.Open "file path"
```

Once the program is open, other scripting commands can be used in order to manipulate the different properties of the program. In order to do this, one would need to use commands of the form

```
Word.Selection.property
```

The various Word properties that one has access to include

```
TypeText "string" – This property outputs the string to the program.
TypeParagraph – This executes a carriage return.
Font.Name = "string" – This changes the type of font.
Font.effect = Boolean – Changes the appearance of the font.
```

The 'effect' properties include Strikethrough, Bold, SmallCaps, Underline, etc., ... as specified in the Word font dialogue. These effects are turned on by setting the Boolean value to True and turned off by setting the value to False.

The 'CreateObject' command is used in conjunction with the program ID (In the above example, the program ID is Word.Application) in order to open a program. This function call returns control of an instance of the program to the script. The 'CreateObject' command is universal in order to open other software. However, once the program application is open, each program will respond to a different set of commands.

	Microsoft Word	Microsoft Excel	Infolytica product
<b>Opening the Application</b>	<b>Set Name = CreateObject</b> ("Word.Application")	<b>Set Name = CreateObject</b> ("Excel.Application")	<b>Set Name = CreateObject</b> ("MagNet.Application")
<b>Making the Application Visible</b>	Name.visible = True	Name.visible = True	Name.visible = True
<b>Opening a New File</b>	Name.documents.add	Name.workbooks.add	Call Name.NewDocument
<b>Opening an Existing File</b>	Name.documents.open "file path"	Name.workbooks.open "file path"	Call Name.OpenDocument ("file path")

#### ActiveX scripting commands

Writing programs to access any of the three Infolytica products (MagNet/ElecNet/ThermNet), or any component of the Microsoft Office suite, is relatively easy due to the ability to record user script in Infolytica products, and the ability to record macros in Microsoft Office.

Macros in Word and Excel are similar to Infolytica's user scripts, in that they are recorded by the application and are written in Visual Basic format. In order to record a macro, one would click on the record macro option in the 'Tools' menu and then execute a series of commands within the program using the keyboard and/or mouse. One would then click on the 'Stop' button in the macro toolbar in order to stop recording the macro. The recorded macros can then be viewed by clicking on the Visual Basic Editor (Tools ⇒ Macro ⇒ Visual Basic Editor). In the editor window, the Visual Basic commands that are used to execute the commands within the program can be viewed. These scripting commands can, in turn, be used within an Infolytica script, or one can edit the script in order to access the application. By recording, viewing and editing macros, one can then learn a range of commands needed to access the various features of the program.

It is important to note that there are some structures that are available in the Visual Basic for Applications Editor in Word and Excel which are not available in VBScript. Therefore, not all of the macro commands can be directly transferred into an Infolytica script. However, they are highly similar and should be easily recognizable once the VBScript notation becomes familiar.

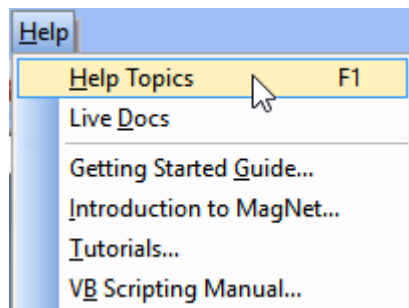
Macros recorded within Excel or Word can be edited such that they could access MagNet, ElecNet or ThermNet. One can create, solve and extract data from a model and import the data into the new application. One can access Infolytica scripting commands from other programs by appending the command after the name given to the Infolytica application (represented as 'Name' in the table above). For example, in order to draw a line in MagNet from within another application, the following function call could be used where 'Name' is a handle to the MagNet application:

```
Call Name.GetDocument.getView.NewLine(0,0,1,1)
```

(Note that the 'GetDocument' portions of the scripting command refers to the model that is currently open in MagNet, either a new model or an existing one. This command is not used in order to open a new model file.)

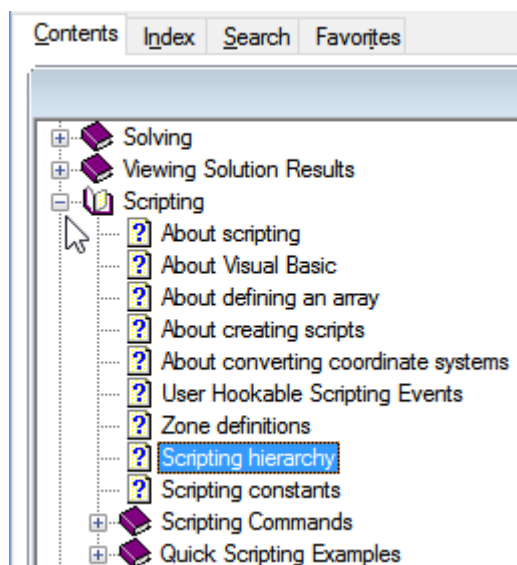
# Chapter 5: Tips for Scripting With Infolytica Products

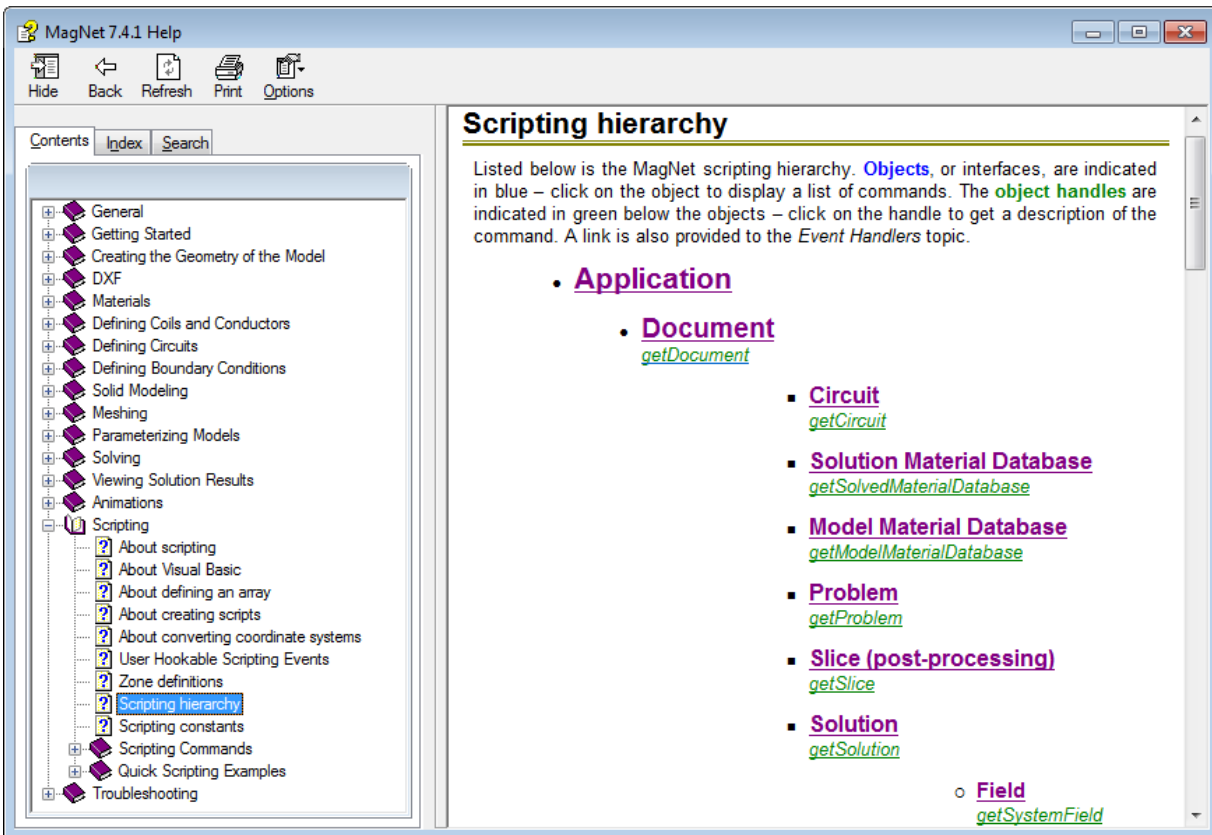
Information on the individual scripting commands used in this chapter can be found in the on-line help (i.e. Help Topics). Note that the syntax for the various commands listed below is subject to change without notice. The updated scripting syntax listed in the on-line help of the Infolytica products should be used at all times.



## SCRIPTING INTERFACES

An interface is a collection of logically related scripting functions. In the on-line help, opening the Scripting chapter and then clicking on the 'Scripting Hierarchy' button gives a list of all available interfaces.





Partial Scripting Hierarchy of MagNet

The main interface in all Infolytica applications is the 'Application' interface. Other interfaces found in the application are subsets of this main one. In order to access a given interface, interfaces are chained together in a script command. For example, to access the field interface in the hierarchy above, one would use the command:

```
Set Field = getDocument.getSolution.getSystemField(...)
```

The interface of all parent interfaces must be included when accessing a given interface. However, handles to the various interfaces can be created using the 'Set' command. One can create the field object listed above as follows:

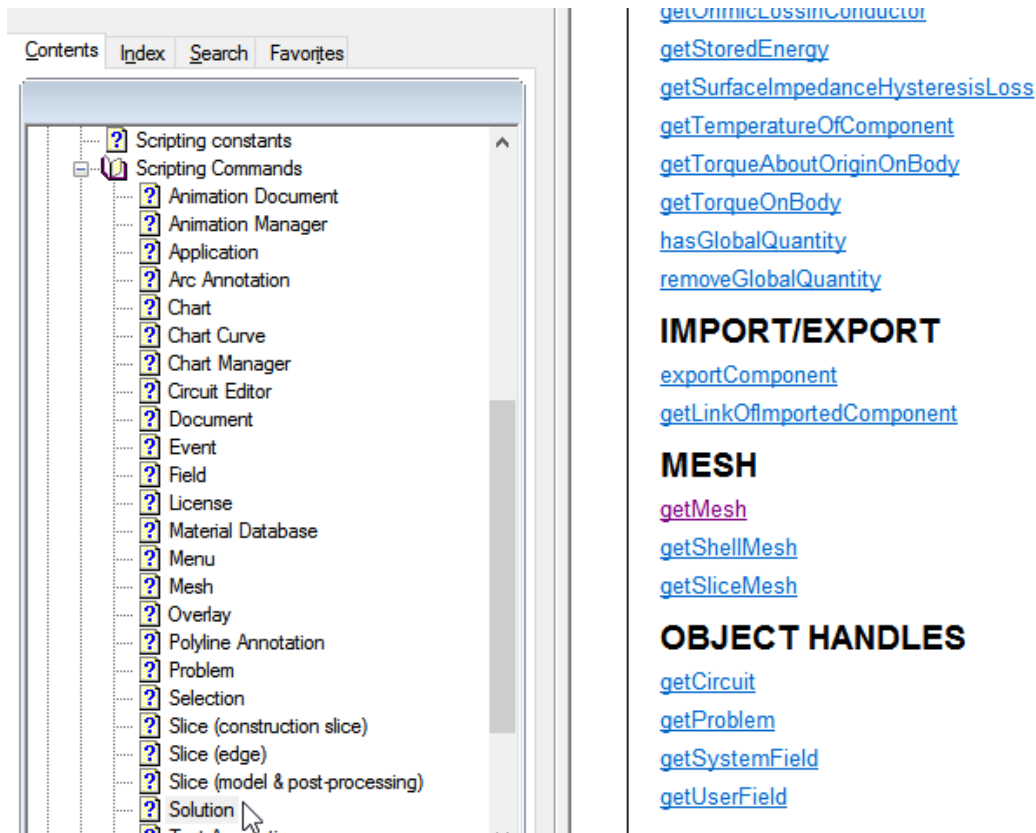
```
Set Document = getDocument
Set Solution = Document.getSolution
Set Mesh = Solution.getMesh(1)
Set Field = Solution.getSystemField(Mesh, "|B|")
```

In this way, one can avoid creating long interface chains when scripting. Note that the Set command must be used when creating a handle to a scripting interface.

Notice that in order to get the Field using the getSystemField command, we must first access the 'Solution' interface. Although the 'Mesh' interface does not need to be accessed when calling the command, it is required as an input argument so one must also get a handle to the 'Mesh' interface.

## SCRIPTING COMMANDS

To get info about a scripting command like `getMesh` which is associated with the 'Solution' interface, open the Help Topics then click on 'Solution' (via the Scripting Hierarchy page or in the 'Scripting Commands' chapter) then click on `getMesh` to access its help page.



### **Solution::getMesh**

#### DESCRIPTION

Returns the 2d or 3d-tet mesh of the requested solution.

**Note** Specifying the zone definition when getting the mesh is more efficient than getting the full mesh and then zoning the mesh.

#### SYNTAX

`getMesh(Solution Id, Zone Definition)`

#### INPUT ARGUMENTS

##### **Solution Id**

An integer that describes the problem ID of a solved problem. The problem ID must be greater than or equal to 1.

or

An array containing the problem ID and time instant for the transient solutions. If time isn't given, then first time solution is used.

**Note** The time instant is in the time unit that was in effect during the solve (e.g. "ms").

If a model file has no solution then obviously, a field cannot be obtained. By referring to MagNet's Help Topics on Scripting we can find the command we need to solve a model. Commands that invoke a solver are associated with the 'Document' interface. For example, one can invoke the 2D static solver using the command:

```
Call getDocument().solveStatic2D()
```

Many other scripting commands can be found in the Help Topics. They can also be found by manually performing the operations and looking at the session log or recording a script as described in *Chapter 1*. One can also find more info by searching for the name of the scripting command.

## PASSING ARRAYS AS INPUT VALUES

Often, when arrays are passed to different function calls in MagNet, ElecNet and ThermNet, they are resized by the function and filled with data. As a result, arrays that are passed should always be declared as a variant and then cast as an array using the ReDim command.

```
Dim ArrayName
ReDim ArrayName(0)
Set Solution = getDocument.getSolution
Set Mesh = Solution.getMesh(1)
Set Field = Solution.getSystemField(Mesh, "|B|")
' The application is able to resize the array
' and places the value of the field at the
' specified point in the array ArrayName
Call Field.getFieldAtPoint(0,0,0, ArrayName)
```

Since there is little extra overhead when creating a variant and then using the ReDim command to create an array, it is recommended that all scripts that call Infolytica APIs define the arrays in this way.

## INFOLYTICA SPECIFIC EVENT HANDLERS

Event handlers are script files that contain code which are automatically triggered once a given event occurs in MagNet, ElecNet or ThermNet. These events are linked to the three main interfaces – Application, Document and View. Event handlers are powerful tools that can be used to automatically configure and customize the graphical user interface. For example, the following script excerpt automatically changes the default color of construction lines every time a model file (i.e. document) is opened or a new one is created.

```
Sub Document_OnLoad ' This subroutine is called every time a document is opened
' Changes the default color of construction slice lines & arcs to orange using RGB values
Call getDocument().setPen(14, infoSolidPen, 1, 255, 128, 128)
End Sub
```

By customizing of this event handler one can avoid doing this operation manually every time a document is loaded.

Event Handlers can be used in diagnostics, modeling and post-processing among other utilities. In the *Tools* section of Infolytica's OCSC website (<http://www.infolytica.com/en/support/>), several different examples of Event Handler files can be downloaded and used.



## IMPORTANT INFOLYTICA APIs FOR ACCESSING THIRD PARTY SOFTWARE<sup>1</sup>

Although languages in the Visual Basic family are often used in ActiveX applications, such as MagNet and Excel, this is not always the case. One example is MATLAB®. This ActiveX compliant software runs its own scripting language. As a result, the way arrays and input parameters are handled is different than the way they are in Infolytica programs. Therefore, some of the Infolytica APIs cannot be called directly from within MATLAB®. However, there are specific APIs in Infolytica applications that can be used as workarounds.

### ProcessCommand API

There are several APIs in MagNet that pass parameters by reference instead of by value. There are some applications which do not support the passing of parameters by reference and therefore, the commands cannot be called directly from within their interface. The processCommand API is a powerful scripting API included with the Infolytica package that can navigate around this problem.

The processCommand API is a member of the application interface. This API takes in a string and executes the string as a VBScript command. The two snippets of code written below behave in the same manner when executed from within an Infolytica application.

```
Dim X, Y, Z
X = 4
Y = 5
Z = X * Y
Call MsgBox (CStr(Z))
```

Running VBScript Code

```
Call processCommand("Dim X, Y, Z")
Call processCommand("X = 4")
Call processCommand("Y = 5")
Call processCommand("Z = X * Y")
Call processCommand("Call MsgBox (CStr(Z))")
```

Using the processCommand API to execute the same VBScript Code

Note that when using the processCommand statement, the values assigned to the variable remains in memory, such that Z is assigned the value 20.

Using the MATLAB® application as an example and MagNet as a remote server, the above code could be executed using the following command sequence (note that in MATLAB®, strings are enclosed in single quotes<sup>2</sup>).

```
MagNet = actxserver('MagNet.Application');           % Create a handle to the MagNet application
Set (MagNet, 'Visible', 'True');                     % Make the application visible
Invoke(MagNet, 'processCommand', 'Dim X, Y, Z');
Invoke(MagNet, 'processCommand', 'X = 4');
Invoke(MagNet, 'processCommand', 'Y = 5');
Invoke(MagNet, 'processCommand', 'Z = X * Y');
Invoke(MagNet, 'processCommand', 'Call MsgBox(CStr(Z))');
```

<sup>1</sup> All Infolytica APIs listed in this section are universal to MagNet, ElecNet and ThermNet. Additional information on the syntax of the scripting APIs mentioned in this section can be found in the help of the various applications.

<sup>2</sup> For more information on the MATLAB® syntax, please refer to the MATLAB® documentation or refer to the MathWorks website at <http://www.mathworks.com>

This workaround allows the power of both applications to be harnessed with a single script. In this way, one can call APIs that contain parameters that are passed by reference which otherwise could not be executed from MATLAB®.

## GetVariant and SetVariant APIs

In VBScript, all variables are defined as variants and have the potential to hold any type of values. However, each programming language has its own methods for dealing with variables, which may not be compatible with VBScript. Furthermore, some applications may not allow specific types to be passed as parameters; this is especially true in the handling of arrays. In order to send arrays back and forth between an Infolytica application and another, the getVariant and setVariant commands can be used. These two commands save and retrieve values from a variant array that is global to the entire program. Values calculated within one application or the other can be stored in this global array and then the value can be transferred to another application.

The getVariant API takes two input values – an index describing where the value should be stored or retrieved from, and a string indicating the category of the variants. The setVariant API takes an additional input, the value to be stored in this global array. For example, the following code stores two different values in the global Variant array. Both inputs are stored at index 0, however, they have two different categories, one labeled ‘StringInput’ and the other labeled ‘NumericalInput’

```
Call setVariant(0, "Value", "StringInput")  
Call setVariant(0, 4, "NumericalInput")
```

In this way, values that are calculated using the processCommand API can then be stored in the Variant array can then be retrieved using the getVariant API.

## RunScript API

The runScript API can take a text file and run it as though it were a .vbs script file. The API takes a file path as an input and executes the named file. With this functionality, values can be calculated or manipulated in one application, and then based on these calculations a text file containing Infolytica API and VBScript commands can be written out. The file can then be executed using runScript.

# Appendix A

## JSCRIPT

In MagNet, ElecNet and ThermNet, using JScript commands and syntax, instead of VBScript is an option. All that must be done to effect the change in programming language is to change the Active Engine of the script from VBScript to JScript.

Most of the commands and structures for VBScript have analogous structures in JScript although the exact syntax differs. Furthermore, the structure for writing subs, functions and event handlers is different. However, while the syntax for the statements may be different, the programming principles detailed in this manual still apply. Note that the program session log and the user session log are recorded exclusively in VBScript.

## REFERENCES

There are many programming references on VBScript on the internet and in print. The most up to date information is found on the Microsoft's website at (<http://msdn.microsoft.com/en-us/library/d1wf56tt.aspx>). This site contains examples and references on how to program in VBScript.

For more information on the different commands, references for Microsoft's Visual Basic can be used. VBScript is a subset of Visual Basic and as such, most commands are identical. However, some of the commands for Visual Basic cannot be used in VBScript.

For examples on how to script specifically in MagNet, ElecNet and ThermNet, please refer to the **Live Docs**, our web-based documentation system which can be found on Infolytica's Online Customer Support Center (OCSC) website (<http://www.infolytica.com/en/support/>). Almost every section of Live Docs has a Tools section which contains custom scripts. There is also a Scripting section which contains more code examples and additional explanations. The script commands in the scripts are readily accessible and can be modified or used in the generation of new scripts. In addition, there are Excel spreadsheets and Matlab® programs that demonstrate software interaction. In order to access Live Docs, you must have an active maintenance contract which has an associated OCSC username and password. To obtain this info, please contact your sales representative ([sales@infolytica.com](mailto:sales@infolytica.com)).

## SCRIPT FORMS

Script forms are no longer supported in Infolytica products. This is due to Microsoft discontinuing support for them. Please see Release notes for more information at: [http://www.infolytica.com/secured/customer/downloads/release-notes/magnet7.3.aspx#VBScript forms](http://www.infolytica.com/secured/customer/downloads/release-notes/magnet7.3.aspx#VBScript%20forms)

The latest version of the End-User License Agreement is available on-line at:  
[www.mentor.com/eula](http://www.mentor.com/eula)

**IMPORTANT INFORMATION**

**USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.**

**END-USER LICENSE AGREEMENT ("Agreement")**

This is a legal agreement concerning the use of Software (as defined in Section GRA) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

**1. ORDERS, FEES AND PAYMENT.**

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2; (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if

Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

### **3. BETA CODE.**

- 3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

### **4. RESTRICTIONS ON USE.**

- 4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer's product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
- 4.3. Customer agrees that it will not subject any Product to any open source software ("OSS") license that conflicts with this Agreement or that does not otherwise apply to such Product.
- 4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 4.5. The provisions of this Section 4 shall survive the termination of this Agreement.

5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.
6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations ("Third Party Terms") may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement (including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.
7. **LIMITED WARRANTY.**
- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
- 7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
9. **THIRD PARTY CLAIMS.**
- 9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.
- 9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.
- 9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

## 10. INFRINGEMENT.

- 10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
- 10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

## 11. TERMINATION AND EFFECT OF TERMINATION.

- 11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is

revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.

16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.