Enable Basic Reference

Summary

This reference manual describes how to use Enable Basic language to create scripts.

The following topics in this Enable Basic Language Reference provide an overview of the structure of a script written in the Enable Basic language:

Exploring the Enable Basic Language

The Enable Basic language follows similar conventions to Microsoft Visual Basic. Before attempting to create scripts using Enable Basic, you should be familiar with the process-based nature of the Altium Designer environment and how parameters are used with processes.

```
'-------------------------------------------------------------------------------
' $Summary Circuit Wizard - Demonstrate placing/connecting parts together.
' Copyright (c) 2004 by Altium Limited
'
' Enable Basic script
'-------------------------------------------------------------------------------

'-------------------------------------------------------------------------------

' Declare your constants here
Const DEFAULT_RES = "200 ohm"
Const DEFAULT_CAP = "100 uf"

'-------------------------------------------------------------------------------
Sub Main
    Dim X As Integer, Y As Integer, R As Integer
    Dim Capacitance As String, Resistance As String

    R = RunDialog(Resistance, Capacitance)
    If  R <> 0 Then
      If GetClickPosition(X, Y) <> 0 Then
         Call PlaceFilter(X, Y, Capacitance, Resistance)
      End IF
    End IF
End Sub
'-------------------------------------------------------------------------------

'-------------------------------------------------------------------------------
Function RunDialog(ByRef Res As String, ByRef Cap As String) As Integer
   ' Dialog box definition
   Begin Dialog DlgType 100,100,180, 50, "Filter Wizard"
         Text             5, 13, 41, 10, "Capacitance:"
         TextBox         54, 12, 50, 12, .Capacitance
         Text             5, 28, 41, 10, "Resistance:"
         TextBox         54, 27, 50, 12, .Resistance
         OKButton       120, 12, 50, 12
         CancelButton   120, 28, 50, 12
   End Dialog

   Dim DlgBox As DlgType
   Dim Result As Integer

   DlgBox.Capacitance = DEFAULT_CAP
   DlgBox.Resistance  = DEFAULT_RES
   Result = Dialog(DlgBox) ' call the dialog box
   Res = DlgBox.Resistance
   Cap = DlgBox.Capacitance
   RunDialog = Result
End Function
```

An Enable Basic script is divided into three main sections:

Global declarations

This is the first section in the script file and contains the declarations of any global variables and constants. Any variables declared using the `Dim` statement in this section are global and can be used by the main program as well as any functions or subroutines.

Main program

This is the main program code. The main program code is defined as a procedure with a `Sub...End Sub` statement block. The name given to the main program block is not important, however it must be the first procedure defined in the script file.

Subroutines and functions

Following the main program block is the definition of any functions or subprocedures which are called by the main program. Each function is defined in a `Function...End Function` statement block and each subprocedure is defined in a `Sub...End Sub` statement block.

EnableBasic Source Files

A script project is organized to store script documents (script units and script forms) which are edited in Altium Designer. You can execute the script from a menu item, toolbar button or from the *Run Script* dialog from the system menu.

PRJSCR and BAS files

Scripts are organized into projects with a `*.PRJSCR` extension. Each project consists of files with a `*.bas` extension.

It is possible to attach scripts to different projects and it is highly recommended to organize scripts into different projects to manage the number of scripts and their procedures / functions.

Scripts consist of functions/procedures that you can call within Altium Designer.

About EnableBasic Examples

The examples that follow illustrate the basic features of Enable Basic programming using simple scripts for the Altium Designer application.

Writing Enable Basic Scripts

In this section we will examine:

- Formatting Values in Enable Basic
- Using Arrays in Enable Basic
- Format for Writing lines of Enable Basic Code
- Enable Basic Naming Conventions
- Using Named Variables in Enable Basic
- Enable Basic Error Codes
- Error Handling in Enable Basic

Formatting Values in Enable Basic

You can format values in an Enable Basic macro script for display using the `Format` function. This function applies a specified formatting template to a value and returns a string which contains the formatted value. The template used to format the value is referred to as the formatting string. The `Format` function takes the form:

```
Format(expression [, fmt$])
```

where

`expression` is any valid Enable Basic expression

`fmt$` is a string or string expression that contains a predefined Enable Basic format name, or a user-defined format string.

Enable Basic provides a number of predefined formats which can be applied by using the format name as the `fmt$` parameter in the `Format` statement.

The power of the `Format` function comes from the ability to accept user-defined formatting strings. These strings use special placeholder characters to define the format of the expression.


For a full description of format strings, see *Creating User-Defined Formatting Strings in Enable Basic.*

Creating User-Defined Formatting Strings in Enable Basic

A user-defined formatting string is used to create a formatting template for use with the `Format` function in an Enable Basic macro script. For example, the following code fragment uses the formatting string "###0.00" to format a number with at least one leading digit and two decimal places:

```
Format(334.9, "###0.00")
```

This format statement returns the string "334.90". The "#" and "0" characters have special meanings in a formatting string. The following are lists of the special characters for use in formatting strings and their meanings.

Special Characters Used in Number Format Strings

The following characters are used to define number formatting strings:

Character	Meaning
0	Digit placeholder. If there is no corresponding digit in the format expression, leading and/or trailing zeroes will be added. When used to the left of the decimal placeholder, indicates the number of decimal places displayed.
#	Optional Digit placeholder. If there is a digit in the corresponding placeholder position, it is displayed. Otherwise no digit is displayed.

Character	Meaning
.	Decimal placeholder.
%	Percentage placeholder. The percent character (%) is inserted in the position where it appears in the format string. When this character is used, the expression is multiplied by 100.
,	Thousand separator. Indicates that a comma is to be inserted between each group of three digits to the left of the decimal point. The character must have a digit placeholders (0 or #) on either side of it in the format string.
E+, e+ E-, e-	Scientific format. If the format expression contains at least one digit placeholder (0 or #) to the right of E-,E+,e- or e+, the number is displayed with "E" or "e" inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E- or e- to place a minus sign next to negative exponents. Use E+ or e+ to place a plus sign next to positive exponents.
\	Interprets the next character in the format string as a literal. The backslash itself isn't displayed. Use this to display the special formatting characters as text in the output format. To display a backslash, use two backslashes (\\).
*	Display the next character as the fill character. Any empty space in a field is filled with the character following the asterisk.

Special Characters Used in Date & Time Format Strings

The following characters are used to define date and time formatting strings:

Character	Meaning
/	Date separator. The actual character used as the date separator in the formatted out depends on the system date format.
:	Time separator. The actual character used as the time separator depends on the system time format.
C	Displays the date as dddd and displays the time as ttttt.
d	Display the day of the month as a number without a leading zero (1-31)
dd	Display the day of the month as a number with a leading zero (01-31)
ddd	Display the day of the week as an abbreviation (Sun-Sat).
ddddd	Display a date serial number as a complete date (including day, month, and year).
w	Display the day of the week as a number (1- 7).
ww	Display the week of the year as a number (1-53).
m	Display the month as a number without a leading zero (1-12). If m immediately follows h or hh, the minute rather than the month is displayed.
mm	Display the month as a number with a leading zero (01-12). If mm immediately follows h or hh, the minute rather than the month is displayed.
mmm	Display the month as an abbreviation (Jan-Dec).
mmmm	Display the month as a full month name (January-December).
q	Display the quarter of the year as a number (1-4).
y	Display the day of the year as a number (1-366).
yy	Display the day of the year as a two-digit number (00-99)
yyyy	Display the day of the year as a four-digit number (100-9999).
h	Display the hour as a number without leading zeros (0-23).

Character	Meaning
hh	Display the hour as a number with leading zeros (00-23).
n	Display the minute as a number without leading zeros (0-59).
nn	Display the minute as a number with leading zeros (00-59).
s	Display the second as a number without leading zeros (0-59).
ss	Display the second as a number with leading zeros (00-59).
ttttt	Display a time serial number as a complete time (including hour, minute, and second) formatted using the system time separator.
AM/PM	Use the 12-hour clock and displays an uppercase AM/PM
am/pm	Use the 12-hour clock and displays a lowercase am/pm
A/P	Use the 12-hour clock and displays a uppercase A/P
a/p	Use the 12-hour clock and displays a lowercase a/p
AMPM	Use the 12-hour clock and displays the system am/pm designators.

Special Characters Used in Text Format Strings

Character	Meaning
@	Character placeholder. Displays a character or a space. Placeholders are filled from right to left unless there is an ! character in the format string.
&	Character placeholder. Display a character or nothing.
<	Force lowercase.
>	Force uppercase.
!	Force placeholders to fill from left to right instead of right to left.

A format string for numbers can have up to three sections separated by semicolons. These sections are: "Default number format[; Negative values][; Zero values]". For example, the following formatting string defines that negative numbers appear in brackets and that zero values return the word "nothing":

```
"#0.00;(#0.00);\n\o\t\h\i\n\g"
```

For 334.9, this returns "334.90". For –334.9, this returns "(334.90)". For 0, this returns "nothing".

A format string for text can have up to two sections: "Default string format[; Null string values]". Date and time format strings have only one section.

The following table gives some examples of formatting strings:

fmt$ =	expr	Format(expr, fmt$)
"#,#0.00"	345655.9	**"345,655.90"**
	-345655.9	**"-345,655.90"**
	0	**"0.00"**
	.3	**"0.30"**
	.003	**"0.00"**
"#,#0.00; -#,#0.00; \N\U\L\L"	345655.9	**"345,655.90"**
	-345655.9	**"-345,655.90"**

fmt$ =	expr	Format(expr, fmt$)
	0	**"NULL"**
	.3	**"0.30"**
	.003	**"0.00"**
"$#,#0.00; ($#,#0.00); \[$0\]"	345655.9	**"$345,655.90"**
	-345655.9	**"($345,655.90)"**
	0	**"[$0]"**
	.3	**"$0.30"**
	.003	**"$0.00"**

Note

The **Format** function always returns a string value, regardless of what data is being formatted.

Using Arrays in Enable Basic

An array is a single named variable that contains a number of indexed elements. Enable Basic supports both single and multi-dimensional arrays. All the elements in an array have the same data type and support is provided for arrays of Integer, Long, Single, Double and String data types. In a multi-dimensional array, each dimension can contain a different number of elements.

Unlike standard variables, you MUST declare an array in a `Dim` or `Static` statement before you can use it. Also, you MUST specify the data type for the array as part of the `Dim` or `Static` statement. Enable Basic does not support dynamic arrays, so once an array is declared you cannot change its size or data type.

When an array is first declared, all elements are empty. To reinitialize all elements in an array after the values have been assigned, use the `Erase` statement.

To reference an array element in a program, you use the following syntax:

```
myArray(i1, i2, i3, ...)
```

where `i1` is the index number of the element in the first dimension, `i2` is the index number of the element in the second dimension, etc.

The Lower Bound of an array dimension is the minimum allowable index value for that dimension. Similarly, the Upper Bound of an array dimension is the largest allowable index value for that dimension. The index numbers for an array must be integers and be within the upper and lower bounds for each particular dimension.

By default, the Lower Bound for all dimensions is 0. You can change the default Lower Bound to 1 by placing an `Option Base 1` statement in the Global Declarations section of a macro script.

Declaration of Arrays in Enable Basic

The general form of an array declaration is as follows:

```
Dim ArrayName(U1, U2, ...) As DataType
```

where

`ArrayName` is the name of the array (which follows standard Enable Basic naming conventions)

`U1`, `U2`... are the upper bounds for each dimension

`DataType` is the type of data to be held by the array.

The following code fragment gives some examples of standard array declarations:

```
Dim Counters(9) As Integer ' single dimension, 10 element array.
Dim myData(4, 9) As Double ' two dim, 50 element (5x10) array.
```

Explicitly declaring the lower bound for an array dimension

The Lower Bound of an array dimension is the minimum allowable index value for that dimension. Similarly, the Upper Bound of an array dimension is the largest allowable index value for that dimension.

Unless the `Option Base 1` statement is used, the default lower bound for each array dimension is 0. You can, however, explicitly define the lower bound for each array dimension by specifying the full index range in the `Dim` statement.

```
Dim ArrayName(L1 To U1, L2 To U2, ...) As DataType
```

where L1, L2... are the lower bounds for each dimension, U1, U2... are the upper bounds for each dimension. The following code fragment illustrates this syntax:

```
Dim Counters (1 To 10) As Integer ' single dimension, 10 element array.
Dim myData(1 To 5, 9) As String   ' two dim, 50 element (5x10) array.
Dim myArray(1 To 5, 11 To 20) As String ' two dim, 50 element (5x10) array.
```

In the preceding declarations, the index numbers of Counters run from 1 to 10, index numbers of myData run from 1 to 5 and 0 to 9 and the index numbers for myArray run from 1 to 5 and 11 to 20.

Format for Writing Lines of Enable Basic Code

Case Sensitivity

Enable Basic is not case sensitive, i.e. all keywords, statements, variable names, function and procedure names can be written without regard to using capital or lower case letters. Both upper and lower case characters are considered equivalent. For example, the variable name myVar is equivalent to myvar and MYVAR. Enable Basic treats all of these names as the same variable.

The only exception to this is in literal strings, such as the title string of a dialog definition, or the value of a string variable. These strings retain case differences.

Statement Separators

In general, each Enable Basic code statement is written on a separate line in the script file. The carriage return character marks the end of a code statement. You can, however, put more than one code statement on a single line by separating the code statements with the colon " **:** " character. For example:

```
X.AddPoint( 25, 100)
X.AddPoint( 0, 75)
```

is equivalent to:

```
X.AddPoint( 25, 100) : X.AddPoint( 0, 75)
```

The Space Character

A space is used to separate keywords in a code statement. However, Enable Basic ignores any additional spaces in a code statement. For example:

```
X = 5
```

is equivalent to

```
X       =           5
```

You may use spaces to make your code more readable.

Splitting a Line of Enable Basic Code

Enable Basic does not put any practical limit on the length of a single line of code in a macro script, however, for the sake of readability and ease of debugging, it is good practice to limit the length of code lines so that they can easily be read on screen or in printed form.

If a line of code is very long, you can break the code by using the "_" underscore continuation character. Place the "_" at the end of a line to indicate that the code continues on the next line. The "_" must be preceded by a space.

The following code fragment shows the use of the continuation character:

```
Declare Sub InvertRect Lib "User" (ByVal hDC As Integer, _
aRect As Rectangle)
```

This code will be treated by the Enable Basic interpreter as if it were written on a single line.

Including Comments in Enable Basic Scripts

In a macro script, comments are non-executable lines of code which are included for the benefit of the programmer. Comments can be included virtually anywhere in a script. Any text following an apostrophe or the word Rem are ignored by Enable Basic.

```
' This whole line is a comment
rem This whole line is a comment
REM This whole line is a comment
```

```
Rem This whole line is a comment
```

Comments can also be included on the same line as executed code. For example, everything after the apostrophe in the following code line is treated as a comment.

```
 MsgBox "Hello"        ' Display Message
```

Enable Basic Naming Conventions

In an Enable Basic macro script file, the main program block and subprocedures are declared using the `Sub...End Sub` statement block. Functions are declared using the `Function...End Function` statement block. Both of these statement blocks require a name to be given to the procedure or function. Also, Enable Basic allows you to create named variables and constants to hold values used in the macro.

In general, there is no restriction to the names you can give to procedures, functions, variables and constants as long as they adhere to the following rules:

- The name can contain the letters A to Z and a to z, the underscore character "_" and the digits 0 to 9.
- The name must begin with a letter.
- The name must be no longer than 40 characters.
- The name cannot be a Enable Basic keyword or reserved word.
- Names are case insensitive when interpreted. You may use both upper and lower case when naming a function, subroutine, variable or constant, however the Enable Basic interpreter will not distinguish between upper and lower case characters. Names which are identical in all but case will be treated as the same name in Enable Basic.

Using Named Variables in Enable Basic

In an Enable Basic script file, you use named variables or constants to store values to be used during program execution. Variables can be used to store a variety of data types.

In Enable Basic you do not need to explicitly declare a variable before using it. You can use a previously undeclared variable name in a code statement and Enable Basic will recognize the variable.

You can explicitly declare a variable to hold a specific data type using the `As Type` clause of the `Dim` or `Static` statements. For example, the following code fragment defines a string variable and an integer variable:

```
Dim myStr As String, myInt As Integer
```

You can implicitly declare the type of data a variable can hold by using one of the Type Specifier characters as the last character of the variable's name. Valid Type Specifiers are:

- **$** = String
- **%** = Integer
- **&** = Long
- **!** = Single
- **#** = Double

For example, the following code fragment also declares a string and an integer variable:

```
Dim myString$, myInt%
```

Variables whose type is not explicitly or implicitly declared are treated as a Variant data type. Variants can store any data type and can vary the data type dependent on the value assigned. To determine the current data type of a Variant, use the `VarType` function. For example, the following code fragment declares a variant and then assigns several different types of data to it:

```
Dim myVar
myVar = 1.2768     ' myVar is a Single data type.
myVar = "Hello"    ' myVar is now a String data type
```

Enable Basic Data Types

The following is a list of data types used by Enable Basic:

Data Type	Type Specifier	VarType =	Size
String	**$**	8	65,500 char
Integer	**%**	2	2 bytes

Long	**&**	3	4 bytes
Single	**!**	4	4 bytes
Double	**#**	5	8 bytes
Variant	-	0	-
Object	-	9	-
Boolean	-	11	-
Date	-	7	-

Data Type Conversion in Enable Basic

In an Enable Basic macro script, named variables used to store program values can be explicitly or implicitly declared to accept a particular data type. If you do not declare the data type for a variable, it is created as a Variant. When you assign a value to a Variant, Enable Basic sets the data type of the variant to suit the value. Assigning a different value to a Variant will change the data type to suit. Use the `VarType` function to return the current variable data type.

When you assign a value to a variable whose data type has been explicitly or implicitly declared, Enable Basic will attempt to automatically convert the value if it is not of the correct data type. For example, the following code fragment declares an Integer variable, myInt and then tries to assign a string value to it.

```
Dim myInt As Integer
myInt = "3.76 amps"    ' myInt is now 4
```

The result of this assignment statement is that Enable Basic will automatically convert the string to an integer and store this integer in myInt. In this case, after the assignment myInt = 4.

Automatic data type conversion means that you do not have to be exact about matching assignment statements and variable types. Be aware, however, that the results of conversion may not always be what you expect, so you should take care that any assignment statements produce predictable results. For example, the code fragment

```
 Dim myInt As Integer
 myInt = "R33"
```

would set myInt = 0 because when Enable Basic converts a string to a number it looks for a numerical value in the string starting from the first character. If the first character is not a number or a decimal point, then the conversion returns 0.

User-Defined Variable Types in Enable Basic

Users can define custom variable types that are composites of other built-in or user-defined types. Variables of these new composite types can be declared and then member variables of the new type can be accessed using dot notation. Variables of user-defined types can not be passed to DLL functions expecting 'C' structures.

User-defined types are created using the `Type...End Type` statement block, which must be placed in the Global Declaration section of your Enable Basic code (all user-defined variable types are global). The variables that are declared as user-defined types can be either global or local. User-defined variable types cannot contain arrays.

User-defined variable types are often used when you need to define a group of linked variables or data records. The following code fragment illustrates this by defining a user-defined variable type called "Component" which includes three elements: Designator, Value and CompType. Together, these elements form the description of a component:

```
Type Component
  Designator As String
  Value As Double
  CompType As String
End Type
```

Once a user-defined type is defined, you can reference individual elements using "dot" notation. For example, the following code fragment declares a variable called “myComp” as a Component type and then sets the value for each element:

```
Dim myComp As Component
myComp.Designator = "R7"
myComp.Value = 1200
myComp.CompType = "Resistor"
```

Enable Basic Error Codes

Err	Description	Err	Description
3	Return without GoSub	**5**	Invalid procedure call
6	Overflow	**7**	Out of memory
9	Subscript out of range	**10**	Array is fixed or temporarily locked
11	Division by zero	**13**	Type mismatch
14	Out of string space	**16**	Expression too complex
17	Can't perform requested operation	**18**	User interrupt occurred
20	Resume without error	**28**	Out of stack space
35	Sub, Function, or Property not defined	**47**	Too many DLL application clients
48	Error in loading DLL	**49**	Bad DLL calling convention
51	Internal error	**52**	Bad file name or number
53	File not found	**54**	Bad file mode
55	File already open	**57**	Device I/O error
58	File already exists	**59**	Bad record length
60	Disk full	**62**	Input past end of file
63	Bad record number	**67**	Too many files
68	Device unavailable	**70**	Permission denied
71	Disk not ready	**74**	Can't rename with different drive
75	Path/File access error	**76**	Path not found
91	Object variable or With block variable not set	**92**	For loop not initialized
93	Invalid pattern string	**94**	Invalid use of Null
429	OLE Automation server cannot create object	**430**	Class doesn't support OLE Automation
432	File name or class name not found during OLE Automation operation	**438**	Object doesn't support this property or method
440	OLE Automation error	**443**	OLE Automation object does not have a default value
445	Object doesn't support this action	**446**	Object doesn't support named arguments
447	Object doesn't support current local setting	**448**	Named argument not found
449	Argument not optional	**450**	Wrong number of arguments
451	Object not a collection	**444**	Method not applicable in this context
452	Invalid ordinal	**453**	Specified DLL function not found
457	Duplicate Key	**460**	Invalid Clipboard format
461	Specified format doesn't match format of data	**480**	Can't create AutoRedraw image
481	Invalid picture	**482**	Printer error
483	Printer driver does not supported specified property	**484**	Problem getting printer information from the system.

Err	Description	Err	Description
485	Invalid picture type	**520**	Can't empty Clipboard
521	Can't open Clipboard		

Error Handling in Enable Basic

In an Enable Basic macro script you can trap and process any run-time errors that occur using the `On Error` statement. This statement defines a jump to a subprocedure that will be run whenever an error is encountered. This statement takes the form:

```
On Error Goto label
```

Where

`label` represents the name of a subprocedure defined in the procedure or function. The error handling subprocedure would then take the form:

```
label:
   ...error handling statement block...
Resume Next
```

The `Resume Next` statement tells the program to resume at the next instruction line after the one that caused the subprocedure call.

Enable Basic maintains an internal object variable called `Err` that stores the error number and description of the last error that occurred. You can use this variable in an error handling subroutine to determine the type of error that has occurred.

The properties of the `Err` object are as follows:

- `Err.Number` returns the number of the last error to occur
- `Err.Description` returns a text description of the last error to occur

Once you have included an `On Error` statement in your procedure, you can temporarily disable error handling by issuing the statement:

```
On Error Goto 0
```

This causes the interpreter to handle errors internally, rather than invoke a user-defined subprocedure. Issue the standard `On Error` statement to re-enable the error handling subprocedure.

Enable Basic Keywords

Reserved words in Enable Basic

The following words are reserved in Enable Basic and cannot be used for variable names.

A, B

Abs, AppActivate, Asc, Atn, Append, As, Base, Beep, Begin, Binary, ByVal

C

Call, Case, ChDir, ChDrive, Chr, Const, Cos, CurDir, Ctype, CDbl, CInt, Clng, Csng, CStrg, Cvar, Close, CheckBox

D, E

Date, Day, Declare, Dim, Dir, Do...Loop, Dialog, DDEInitiate

DDEExecute, End, Exit, Exp

F, G, H

FileCopy, FreeFile, For...Next, Format, Function, GoTo, Global, Hex$, Hour

I, J, K

If...Then...Else...[End If], InputBox, InStr, Int, IsNull, Integer, IsEmpty

IsNull, IsNumeric, IsDate, Kill

L, M, N

LBound, LCase, Left, Len, Let, Log, Ltrim, Mid, MkDir, Month, MsgBox

O, P, Q

Oct, Open, OKButton, Object, Option, Print

R, S, T

Rem, RmDir, Rnd, Return, Rtrim, SendKeys, Set, Second, Select, Shell, Sin, Sqr, Stop, Str, Tan, Text, TextBox, Time, Type, Trim, Trim$ To, Time, Then, Tan

U, V, W, X, Y, Z

UBound, UCase, UCase$, Val, Variant, VarType, Write #

Abs

Description

Returns the absolute value of a number. The data type of the return value is the same as that of the number argument. However, if the number argument is a Variant of VarType String and can be converted to a number, the return value will be a Variant of VarType Double.

Syntax

```
Abs(number)
```

Parameters

number - a variable, constant or expression which returns a valid number.

Example

The following example uses an input box to ask the user to enter a number and then displays the absolute value of the number.

```
Sub Main
 Dim Msg, X, Y
 X = InputBox("Enter a Number:")
 Y = Abs(X)
 Msg = "The number you entered is "  & X
 Msg = Msg & ". The Absolute value of " & X & " is " & Y
 MsgBox Msg                          ' Display Message.
End Sub
```

AppActivate

Description

Activates a currently open application.

Syntax

```
AppActivate ApplicationName$
```

Parameters

ApplicationName$ - a string or string expression that resolves to the name that appears in the title bar of the application window to activate.

Notes

The AppActivate procedure is only applicable to applications which are currently running. To run a program, use the Shell function.

Example

This example changes the Windows Calculator mode.

```
Sub Main ()
 Ent = Chr(32)                  ' Define the enter key
 X = Shell("Calc.exe", 1)       ' Run the  Calculator
 MsgBox "Change to Scientific mode"
 AppActivate "Calculator"       ' Ensure Calc visible
 SendKeys "%" + Ent + "R", True
 SendKeys "%VS", True           ' Activate scientific
 MsgBox "Change to Standard mode"
 AppActivate "Calculator"       ' Focus Calculator
 SendKeys "%VT", True           ' Activate standard
```

```
End Sub
```

Atn

Description

Returns the arc tangent of a number.

Syntax

```
Atn(number)
```

Parameters

number - a numeric variable, constant or expression.

Notes

The return result is a numeric value representing the arc tangent of number expressed in radians. The function will by default return a Double data type unless a Single or Integer is specified as the return value.

Example

The following example displays a message dialog showing the value of Pi.

```
Sub AtnExample ()
 Dim Msg, Pi               ' Declare variables.
 Pi = 4 * Atn(1)           ' Calculate Pi.
 Msg = "Pi is equal to " & Str(Pi)
 MsgBox Msg                ' Display results.
End Sub
```

Asc

Description

Returns a numeric value that is the ASCII code for the first character in String$.

Syntax

```
Asc(String$)
```

Parameters

String$ - a character string or string expression.

Notes

If String$ contains more than one character, Asc only returns the ASCII code for the first character in the string. To convert an ASCII code to its corresponding character, use the Chr function.

Example

The following example displays a message dialog containing all the letters of the alphabet.

```
Sub Main ()
 Dim I, Msg                  ' Declare variables.
 For I = Asc("A") To Asc("Z") ' From A through Z.
  Msg = Msg & Chr(I)         ' Create a string.
 Next I
 MsgBox Msg                  ' Display results.
End Sub
```

Beep

Description

Sounds a tone through the computer's speaker.

Syntax

```
Beep
```

Parameters

This procedure accepts no parameters.

Notes

The frequency and duration of the beep depends on hardware, which may vary among computers.

Example

The following example prompts the user to enter a number between 1 and 3. If the user enters a number out of this range, the computer will beep and display a message.

```
Sub BeepExample ()
 Dim Answer, Msg              ' Declare variables.
 Do
  Answer = InputBox("Enter a value from 1 to 3.")
  If Answer >= 1 And Answer <= 3 Then ' Check range.
   Exit Do                    ' Exit Do...Loop.
  Else
   Beep                       ' Beep if not in range.
  End If
 Loop
 MsgBox "You entered a value in the proper range."
End Sub
```

Call

Description

Activates an Enable Basic subroutine called SubName or a DLL function within the dynamic linked library called SubName.

Syntax

```
[Call] SubName [(parameter list)]
```

Parameters

SubName - the name of the function or procedure to call.

parameter list - a comma-delimited list of arguments to pass to the called function or procedure.

Notes

Use of the Call keyword is optional when calling an Enable Basic subprocedure, user-define function or a DLL function, and is only included for compatibility with some older versions of Enable Basic.

Note: Parentheses must always be used to enclose the argument list if the Call syntax is being used.

Example

This example uses the Call statement to the DisplayMsg subprocedure and then calls the same procedure without the Call statement. Note that when the Call statement is used, the parameter list for a subprocedure must be enclosed in parentheses.

```
Sub Main ()
 Dim Msg As String
 Msg = "Answering a call"
 Call DisplayMsg(Msg)      ' Using the Call statement.
 DisplayMsg Msg            ' Not using the Call statement.
End Sub
Sub DisplayMsg(ByVal myString As String)
 MsgBox myString
End Sub
```

CBool

Description

Converts expression to a Boolean TRUE or FALSE

Syntax

```
CBool(expression)
```

Parameters

expression - a valid numeric expression.

Notes

The function returns a Boolean data type, which is either TRUE or FALSE.

If expression is numeric, the function returns FALSE if expression = 0, or TRUE if expression = any non-zero number.

If expression is a string, the function converts the string to a numeric value before conversion to a Boolean.

Example

The following example shows how different expressions are converted to a Boolean TRUE or FALSE.

```
Sub Main
 Print CBool(0)          ' Returns FALSE
 Print CBool(1)          ' Returns TRUE
 Print CBool(345.778)    ' Returns TRUE
 Print CBool(-255)       ' Returns TRUE
 Print CBool("Hello")    ' Returns FALSE
 Print CBool("0 volts") ' Returns FALSE
 Print CBool("5 volts") ' Returns TRUE
End Sub
```

CDbl

Description

Converts expression to a Double data type. The Double data type represents a double-precision floating point number.

Syntax

```
CDbl(expression)
```

Parameters

expression - a valid string or numeric expression.

Example

This example converts a single precision integer y to a double precision number x.

```
Sub Main ()
 Dim y As Integer
 y = 25
 If  VarType(y) = 2 Then
  Print y
  x = CDbl(y)    ' Converts the integer value of y to
                 ' a double value in x
  Print VarType(x)
 End If
End Sub
```

CInt

Description

Converts any valid expression to an integer.

Syntax

```
CInt(expression)
```

Parameters

expression - any valid expression.

Notes

The function returns an Integer data type derived from the expression parameter. If expression is a floating point number, then the return value is the rounded integer value of the number. If expression is a string, the return value is 0, except when the string starts with a number, in which case the return value is the rounded value of the number. If expression is a Boolean True, the return value is –1. If expression is a Boolean False, the return value is 0.

Example

The following example shows the return values of the CInt function for various expression types.

```
Sub Main ()
 MsgBox CInt(2.45)       ' Returns 2
 MsgBox CInt(2.55)       ' Returns 3
 MsgBox CInt("Text")     ' Returns 0
 MsgBox CInt("A5")       ' Returns 0
 MsgBox CInt("5A")       ' Returns 5
 MsgBox CInt(True)       ' Returns -1
 MsgBox CInt(False)      ' Returns 0
End Sub
```

CLng

Description

Converts any valid expression into a long integer.

Syntax

```
CLng(expression)
```

Parameters

expression - any valid expression.

Notes

The function returns a Long data type derived from the expression parameter. If expression is a floating point number, then the return value is the rounded integer value of the number. If expression is a string, the return value is 0, except when the string starts with a number, in which case the return value is the rounded integer value of the number. If expression is a Boolean True, the return value is –1. If expression is a Boolean False, the return value is 0.

Example

The following example shows the return values of the CLng function for various expression types.

```
Sub Main ()
 MsgBox CLng(2.45)       ' Returns 2
 MsgBox CLng(2.55)       ' Returns 3
 MsgBox CLng("Text")     ' Returns 0
 MsgBox CLng("A5")       ' Returns 0
 MsgBox CLng("5A")       ' Returns 5
 MsgBox CLng(True)       ' Returns -1
 MsgBox CLng(False)      ' Returns 0
End Sub
```

Const

Description

Assigns a symbolic name to a constant value. A constant must be defined before it is used.

Syntax

```
[Global] Const Name = Expression
```

Parameters

Name - the name of the variable

Expression - an expression that defines the constant.

Notes

Defining a Const outside a procedure or at the module level automatically declares the constant to be global. The use of the Global prefix is not necessary, but is retained for compatibility with earlier versions of Enable Basic. If a constant is defined within a function or procedure, it is local to that function or procedure.

A type declaration character may be used in the Name parameter, however if none is used Enable Basic will automatically assign a data type to the constant: Long (if it is a long or integer), Double (if a decimal place is present), or a String (if it is a string).

Example

In the following example, GloConst and MyConst are defined as global constants (note that outside of a procedure definition Const and Global Const are equivalent) and PI is defined as local to the Main procedure.

```
Global Const GloConst = 142 ' Global to all
Const MyConst = 122         ' procedures in a module
Sub Main ()
 Const PI = 3.14159         ' Local constant
 Myvar = MyConst + PI + GloConst
 Print MyVar
End Sub
```

CreateObject

Description

Creates an OLE automation object.

Syntax

```
CreateObject(class)
```

Parameters

class - a valid OLE class object.

Example

This example creates a letter in Microsoft Word using Word Basic and OLE.

```
Sub Main ()
 Dim word As object
 Set word = CreateObject("Word.Basic")
 word.FileNewDefault
 word.Insert "Dear Sir:"
 word.InsertPara
 word.InsertPara
 word.Insert "The letter you are reading was"
 word.Insert "created with Enable Basic."
 word.InsertPara
End Sub
```

CSng

Description

Converts any valid expression to a Single.

Syntax

```
CSng(expression)
```

Parameters

expression - any valid expression.

Notes

The function returns a single precision number derived from the expression parameter. If expression is a number, then the return value is the single precision value of the number. If expression is a string, the return value is 0, except when the string starts with a number, in which case the return value is the value of the number. If expression is a Boolean TRUE, the return value is –1. If expression is a Boolean FALSE, the return value is 0.

Example

The following example shows the return values of the CSng function for various expression types.

```
Sub Main ()
 MsgBox CSng(2.45)       ' Returns 2.450000
 MsgBox CSng("Text")     ' Returns 0.000000
 MsgBox CSng("A5")       ' Returns 0.000000
 MsgBox CSng("5A")       ' Returns 5.000000
 MsgBox CSng(True)       ' Returns -1.000000
 MsgBox CSng(False)      ' Returns 0.000000
End Sub
```

CStr

Description

Converts any valid expression to a String.

Syntax

CStr(Expression)

Parameters

Expression - any valid expression.

Declare

Description

The Declare procedure makes a reference to an external procedure in a Dynamic Link Library (DLL).

Syntax

```
Declare Sub ProcName Lib LibName$ [Alias AliasName$] ([argument list])
Declare Function ProcName Lib LibName$ [Alias AliasName$] ([argument list]) [As Type]
Declare Function ProcName App [Alias AliasName$] ([argument list]) [As Type]
```

Parameters

ProcName - the name of the function or subroutine being called.

LibName$ - string containing the name of the DLL that contains the procedure.

AliasName$ - string containing the actual procedure name in the DLL, if different from the name specified by the ProcedureName parameter

argument list - Then the optional argument list needs to be passed the format is as follows: ([ByVal] variable [As Type],...)

The optional ByVal keyword specifies that the variable is passed by value instead of by reference.

Type - defines the data type the function returns.

Notes

When declaring a procedure that has no arguments, include empty double parentheses () to assure that no arguments are passed. For example: Declare Sub OnTime Lib "Check" ()

The third version of the syntax listed makes a reference to a function located in the executable file located in the application where Enable Basic is embedded.

Example

The following statement declares a function residing in User.dll called GetFocus that takes no parameters and returns an integer value.

```
Declare Function GetFocus Lib "User" () As Integer
```

The following statement declares a function residing in `User.dll` called `GetWindowText` that takes the parameters hWnd%, Mess$, cbMax%, which are passed as values and returns an integer.

```
Declare Function GetWindowText Lib "User" (ByVal hWnd%, _
ByVal Mess$, ByVal cbMax%) As Integer
```

Dim

Description

The first listed syntax allocates storage for and declares the data type of variables and arrays in a module. The second listed syntax is used to assign an object variable to represent the values of the controls in a user-defined dialog box. The dialog box represented by DlgName must be defined using the Begin Dialog ... End Dialog procedures.

Syntax

```
Dim VarName[(subscripts)] [As Type]
Dim DlgObject As DlgName
```

Parameters

VarName - the name of the variable.

subscripts - subscript dimensions used when dimensioning an array. subscripts must be of the form (Dim1, Dim2, ...), where Dim1, etc are the number of items in each array dimension.

Type - the data type declaration of the variable.

DlgObject - the name of an Object variable to hold the dialog record.

DlgName - the name of the dialog as specified in a dialog definition.

Notes

You can dimension more that one variable in a single Dim statement by separating each variable definition by a comma. If the optional As Type clause is omitted, the variable is defined as a Variant data type.

In Enable Basic, it is not necessary to define a variable with the Dim statement before using it. Using a variable in an assignment-type statement will create the variable.

Using the Dim procedure outside a function or procedure defines variables as Global.

Example

The following statement dimensions variable a as Single:

```
Dim a As double
```

The following statement dimensions **Y** as Integer, **Z** as Single and **S** as a String:

```
Dim Y As Integer, Z As Single, S As String
```

The following statements are equivalent, and dimension **V** as a Variant:

```
Dim V As Variant
Dim V
```

The following statement creates a two-dimensional array of Integers, the first dimension having 3 elements and the second having 10 elements:

```
Dim MyList(2, 9) As Integer
```

The following subprocedure dimensions an object variable, MyDlg, to represent the dialog control values of a user-defined dialog called DialogBox1:

```
Sub Main()
 ' Begin dialog definition
 Begin Dialog DialogBox1 58,60, 161, 65, "Enter your name"
  Text 2,2,64,12, "Type your name:"
  TextBox 6,14,148,12, .nameStr
  OKButton 16,48,40,12
  CancelButton 60,48,36,12
 End Dialog
 ' End of dialog definition
 Dim MyDlg As DialogBox1    ' Dimension object variable for dialog.
 Dialog MyDlg               ' Display the dialog.
 MsgBox MyDlg.nameStr       ' Display the text from the dialog text box.
End Sub
```

IsEmpty

Description

Returns a value that indicates whether a variable has been initialized.

Syntax

```
IsEmpty(VarName)
```

Parameters

VarName - a valid variable name.

Notes

When a numeric or variant type variable is first declared in Enable Basic, it is empty. You can set a variable to be empty using the statement VarName = Empty. The function returns a Boolean TRUE if VarName is empty, otherwise it returns a FALSE.

IsNull

Description

Returns a value that indicates whether a numeric variable contains the NULL value.

Syntax

```
IsNull(VarName)
```

Parameters

VarName - a valid variable name.

Notes

The function returns a Boolean TRUE if VarName contains the NULL value. If IsNull returns a FALSE the variant expression is not NULL. The NULL value is a special value that indicates a variable contains no data. This is different from a null-string, which is a zero length or empty string which has not yet been initialized.

IsNumeric

Description

Indicates if the expression can be converted to a numeric data type.

Syntax

```
IsNumeric(expression)
```

Parameters

expression - any valid Enable Basic expression.

Notes

The function returns a Boolean TRUE if the expression can be resolved to a numeric value, otherwise it returns a false.

Example

The following example checks whether the value entered into an input box is a valid number.

```
Sub Form_Click ()
 Dim TestVar                          ' Declare variable.
 TestVar = InputBox("Enter a number, letter, or symbol.")
 If IsNumeric(TestVar) Then           ' Evaluate variable.
  MsgBox "Entered data is numeric."   ' Message if number.
 Else
  MsgBox "Entered data is not numeric." ' Message if not.
 End If
End Sub
```

Int

Description

Returns the integer portion of a number.

Syntax

```
Int(num)
```

Parameters

num - a number or numeric expression.

Notes

The function does not perform any rounding, but returns an Integer data type which is the integer portion of number.

This function is identical to the Fix function.

InStr

Description

Returns the character position of the first occurrence of SearchString$ within SourceString$.

Syntax

```
InStr(numBegin, SourceString$, SearchString$)
```

Parameters

numBegin - a valid positive integer expression between 1 and 65,535 that sets the starting character for the search.

SourceString$ - a string expression that is the string to be searched.

SearchString$ - a string expression that to be used as the search string.

Notes

The numBegin parameter is NOT optional. Set the parameter to 1 to search the entire SourceString$.

The function returns an Integer data type that is the starting position of SearchString$ within SourceString$. The first character in SourceString$ has a value of 1. If SearchString$ is not found, the function returns 0. The search is case sensitive.

Example

The following example performs two searches on the string "Good Bye". The first looks for "Bye" starting from the first character of the string. The second searches for "Good" starting from the 5th character of the string.

```
Sub Main ()
 B$ = "Good Bye"
 A% = InStr(1, B$, "Bye")    ' Returns 6.
 C% = Instr(5, B$, "Good")   ' Returns 0 as the search is
                             ' is started from the 5th
                             ' character of Good Bye.
 MsgBox A$
 MsgBox B$
End Sub
```

InputBox

Description

Opens a dialog box that allows user input. The dialog contains a prompt string, text entry box, OK and CANCEL buttons.

Syntax

```
InputBox(prompt$[, title$][, default$][, PosX, PosY])
```

Parameters

prompt$ - a string expression displayed in the input box as a prompt to the user.

title$ - a string expression to be displayed in the title bar of the input box.

default$ - a string expression to be used as the default input text.

PosX, PosY - x and y coordinates defining the location of the top left corner of the input dialog box in pixels relative to the top left corner of the Altium Designer window.

Notes

If the user presses the OK button, the function returns a string that entered into the input box by the user. If the user presses the CANCEL button, a null string is returned.

Example

The following example asks the user to enter their name and then displays the name in a message dialog.

```
Sub Main ()
```

```
 myTitle$ = "Greetings"
 myPrompt$ = "What is your name?"
 myDefault$ = ""
 X% = 200
 Y% = 100
 N$ = InputBox(myPrompt$, myTitle$, myDefault$, X%, Y%)
 MsgBox "Hello " & N$
End Sub
```

GetObject

Description

Retrieves an object or object class from a file.

Syntax

```
GetObject(FileName$[, class$])
```

Parameters

FileName$ - a string expression containing the name and path of the file containing the object to retrieve. If FileName$ is an empty string then class is required.

class - a string expression containing the class name of the object to retrieve.

Hex

Description

Hex returns a String data type which represents the hexadecimal number. The hexadecimal value is based on a decimal integer parameter.

Syntax

```
Hex(num)
```

Parameters

num - a number or numeric expression. The parameter is rounded to the nearest integer value before conversion.

Example

Hex conversion example.

```
Sub Main ()
 Dim Msg As String, x%
 x% = 10
 Msg =Str( x%) &  " decimal is "
 Msg = Msg & Hex(x%) & " in hex "
 MsgBox Msg
End Sub
```

Format

Description

Formats a string or number for display using the fmt$ parameter as a template.

Syntax

```
Format(expression [, fmt$])
```

Parameters

expression - any valid expression.

fmt$ - a string or string expression that contains a pre-defined Enable Basic format name, or a user-defined format string.

Notes

This function returns a String data type which contains the value of expression formatted according to the fmt$ parameter. If the fmt$ parameter is omitted or is zero-length and the expression parameter is numeric, Format provides similar functionality to the

Str function by converting the numeric value to a String data type. However, positive numbers converted using Format lack the leading space reserved for displaying the sign of the value, whereas those converted using Str retain the leading space.

Example

This example shows various uses of the **Format** function to format values using both named and user-defined formats. For the date separator (/), time separator (:) and AM/PM literal, the actual formatted output displayed by your system depends on the locale settings on which the code is running. When times and dates are displayed in the development environment, the short time and short date formats of the code locale are used. When displayed by running code, the short time and short date formats of the system locale are used, which may differ from the code locale. For this example, English/United States is assumed.

```
Sub Main
 ' Returns current time in the system short & long time format.
 MsgBox Format(Time, "Short Time")
 MsgBox Format(Time, "Long Time")
 ' Returns current date in the system short & long date format.
 MsgBox Format(Date, "Short Date")
 MsgBox Format(Date, "Long Date")
 '-----------------------------------
 MyTime = "08:04:23 PM"
 MyDate = "January 27, 1993"
 MsgBox Format(MyTime, "h:n:s")           ' Returns "20:4:23"
 MsgBox Format(MyTime, "hh:nn:ss")        ' Returns "20:04:23 "
 MsgBox Format(MyDate, "dddd, mmm d yyyy")' Returns
                                          ' "Wednesday, Jan 27 1993"
                                          '
 ' If a format string is not supplied, a simple string is returned.
 MsgBox Format(23)                       ' Returns "23".
 ' Example number formats
 MsgBox Format(5459.4, "##,##0.00")      ' Returns "5,459.40"
 MsgBox Format(334.9, "###0.00")         ' Returns "334.90"
 MsgBox Format(5, "0.00%")               ' Returns "500.00%"
 ' Example text formats
 MsgBox Format("HELLO", "<")             ' Returns "hello"
 MsgBox Format("This is it", ">")        ' Returns "THIS IS IT"
End Sub
```

Pre-defined formats for fmt$

To use one of the Enable Basic pre-defined formats, include the format name as the fmt$ parameter in a Format statement.

Predefined Numeric Formats

Enable Basic predefined number format names:

fmt$ =	Description
"General"	Display the number as is, with no thousand Separators
"Fixed"	Display at least one digit to the left and two digits to the right of the decimal separator.
"Standard"	Display number with thousand separator and two digits to the right of the decimal separator.
"Percent"	Multiplies the number by 100 and displays it with two decimal places and a percent sign (%) appended to the end.
"Scientific"	Standard scientific notation.
"True/False"	Displays False if number is 0, otherwise displays True.

Predefined Date & Time Formats

Enable Basic predefined date & time format names:

fmt$ =	Description
"Long Date"	Display a Long Date, as defined in the International section of the Control Panel.
"Medium Date"	Display a date in the same form as the Short Date, but with the month shown as abbreviation, rather than a number.
"Short Date"	Display a Short Date, as defined in the International section of the Control Panel.
"Long Time"	Display a Long Time, as defined in the International section of the Control panel. Long Time includes hours, minutes, seconds.
"Medium Time"	Display time in 12-hour format using hours and minutes and the Time AM/PM designator.
"Short Time"	Display time using the 24-hour format (e.g. 17:45)

Creating User-Defined Format Strings for fmt$

To create your own formats, define a string that includes special formatting characters and acts as a "template" for formatting the supplied expression.

A format string for numbers can have up to three sections separated by semicolons. These sections are: "Default number format[; Negative values][; Zero values]"

A format string for text can have up to two sections: "Default string format[; Null string values]"

Date and time format strings have only one section.

To construct each section of the formatting string, use the formatting characters listed below:

Special Characters for Number Format Strings

The following characters are used to define number formatting strings:

Character	Meaning
0	Digit placeholder. If there is no corresponding digit in the format expression, leading and/or trailing zeroes will be added. When used to the left of the decimal placeholder, indicates the number of decimal places displayed.
#	Optional Digit placeholder. If there is a digit in the corresponding placeholder position, it is displayed. Otherwise no digit is displayed.
.	Decimal placeholder.
%	Percentage placeholder. The percent character (%) is inserted in the position where it appears in the format string. When this character is used, the expression is multiplied by 100.
,	Thousand separator. Indicates that a comma is to be inserted between each group of three digits to the left of the decimal point. The character must have a digit placeholders (0 or #) on either side of it in the format string.
E+, e+ **E-, e-**	Scientific format. If the format expression contains at least one digit placeholder (0 or #) to the right of E-,E+,e- or e+, the number is displayed with "E" or "e" inserted between the number and its exponent. The number of digit placeholders to the right determines the number of digits in the exponent. Use E- or e- to place a minus sign next to negative exponents. Use E+ or e+ to place a plus sign next to positive exponents.
****	Interprets the next character in the format string as a literal. The backslash itself isn't displayed. Use this to display the special formatting characters as text in the output format. To display a backslash, use two backslashes (\\).
*****	Display the next character as the fill character. Any empty space in a field is filled with the character following the asterisk.

Special Characters for Date & Time Format Strings

The following characters are used to define date and time formatting strings:

Character	Meaning
/	Date separator. The actual character used as the date separator in the formatted out depends on the system date format.
:	Time separator. The actual character used as the time separator depends on the system time format.
c	Displays the date as dddd and displays the time as ttttt.
d	Display the day of the month as a number without a leading zero (1-31)
dd	Display the day of the month as a number with a leading zero (01-31)
ddd	Display the day of the week as an abbreviation (Sun-Sat).
ddddd	Display a date serial number as a complete date (including day , month, and year).
w	Display the day of the week as a number (1- 7).
ww	Display the week of the year as a number (1-53).
m	Display the month as a number without a leading zero (1-12). If m immediately follows h or hh, the minute rather than the month is displayed.
mm	Display the month as a number with a leading zero (01-12). If mm immediately follows h or hh, the minute rather than the month is displayed.
mmm	Display the month as an abbreviation (Jan-Dec).
mmmm	Display the month as a full month name (January-December).
q	display the quarter of the year as a number (1-4).
y	Display the day of the year as a number (1-366).
yy	Display the day of the year as a two-digit number (00-99)
yyyy	Display the day of the year as a four-digit number (100-9999).
h	Display the hour as a number without leading zeros (0-23).
hh	Display the hour as a number with leading zeros (00-23).
n	Display the minute as a number without leading zeros (0-59).
nn	Display the minute as a number with leading zeros (00-59).
s	Display the second as a number without leading zeros (0-59).
ss	Display the second as a number with leading zeros (00-59).
ttttt	Display a time serial number as a complete time (including hour, minute, and second) formatted using the system time separator.
AM/PM	Use the 12-hour clock and displays an uppercase AM/PM
am/pm	Use the 12-hour clock and displays a lowercase am/pm
A/P	Use the 12-hour clock and displays a uppercase A/P
a/p	Use the 12-hour clock and displays a lowercase a/p
AMPM	Use the 12-hour clock and displays the system am/pm designators.

Special Characters for Text Format Strings

Character	Meaning
@	Character placeholder. Displays a character or a space. Placeholders are filled from right to left unless there is an ! character in the format string.
&	Character placeholder. Display a character or nothing.
<	Force lowercase.
>	Force uppercase.
!	Force placeholders to fill from left to right instead of right to left.

Fix

Description

Returns the integer portion of a number.

Syntax

```
Fix(number)
```

Parameters

number - a number or numeric expression.

Notes

This function does not perform any rounding, it returns the integer portion of the number.

This function is identical to the Int function.

LBound

Description

Returns the smallest available subscript for the dimension of the indicated array.

Syntax

```
LBound(array [, dimension])
```

Parameters

array - a valid array name, excluding parentheses.

dimesion - a optional numeric expression representing an array dimension of a multi-dimensional array. The first dimension of an array is 1, the second 2, etc.

Notes

If dimension is omitted and array has more than one dimension, the lower bound for the first dimension is returned.

Example

The following example displays the upper and lower bounds for each dimension of a two-dimensional array.

```
Sub Main()
 Dim myArray(2 To 6, 4 To 8)
 Dim Low1, Low2, Up1, Up2, Msg
 Low1 = LBound(myArray)
 Up1 = UBound(myArray)
 Low2 = LBound(myArray, 2)
 Up2 = UBound(myArray, 2)
 MsgBox "First dimension goes from element " & Low1 & " to " & Up1
 MsgBox "The 2nd dimension goes from element " & Low2 & " to " & Up2
End Sub
```

Let

Description

Assigns a value to a variable.

Syntax

```
[Let] VarName = expression
```

Parameters

VarName - a valid variable name.

expression - any Enable Basic expression.

Notes

Let is an optional keyword which is retained for compatibility with older versions of Enable Basic.

MsgBox

Description

Displays a message in a dialog box and waits for the user to click a button on this dialog.

Syntax

```
MsgBox Msg$, [TypeNum] [, Title$]
Var = MsgBox(Msg$, [TypeNum] [, Title$])
```

Parameters

Msg$ - a string expression which gives the message to display in the message box.

TypeNum - an optional numeric expression that defines the buttons, icons and behavior of the message box. For a list of valid numbers, see below. The default is 0.

Title$ - an optional string expression that defines the text to appear in the title bar of the message box.

Notes

The function form of MsgBox returns an Integer data type indicating which button the user presses:

1 = OK; 2 = Cancel; 3 = Abort; 4 = Retry; 5 = Ignore; 6 = Yes; 7 = No

When using the procedure form of MsgBox, you can display the various buttons by setting TypeNum, but there is no way to determine which button the user presses.

Example

This example displays a series of message dialogs with different button combinations. When the user clicks a button, the program displays a message dialog with an information icon telling the user which button was pressed.

```
Sub Main
 For myCount = 1 to 5
  Msg$ = "TypeNum = " & myCount & Chr(10) & "Select a button"
  ButNum = MsgBox(Msg$, myCount, "Test") ' Message box function
  Select Case ButNum
   Case 1
    ButName$ = "OK"
   Case 2
    ButName$ = "Cancel"
   Case 3
    ButName$ = "Abort"
   Case 4
    ButName$ = "Retry"
   Case 5
    ButName$ = "Ignore"
   Case 6
    ButName$ = "Yes"
   Case 7
    ButName$ = "No"
```

```
  End Select
  Msg$ = "You selected the <" & ButName$ & "> button."
  MsgBox Msg$, 64, "Button"     ' Message box procedure with info icon
 Next myCount
End Sub
```

Determining the value for TypeNum

By setting a value for the optional TypeNum parameter, you can control the look and functioning of the message box. The value is arrived at by selecting an option in each of the following areas: buttons displayed, icon displayed, default button and dialog mode. Each option has a key number associated with it. The final value of TypeNum is calculated by adding together each key number for the options chosen.

Example

The following table lists key values for the TypeNum parameter. To calculate a value for TypeNum, select one option from each category and add together the corresponding key numbers.

Buttons

- **0** Display **OK** button only.
- **1** Display **OK** and **Cancel** buttons.
- **2** Display **Abort**, **Retry**, and **Ignore** buttons.
- **3** Display **Yes**, **No**, and **Cancel** buttons.
- **4** Display **Yes** and **No** buttons.
- **5** Display **Retry** and **Cancel** buttons.

Icons

- **16** Display warning icon.
- **32** Display question mark icon.
- **48** Display exclamation mark icon.
- **64** Display information icon.

Default button

- **0** First button is default.
- **256** Second button is default.
- **512** Third button is default.

Mode

- **0 Application modal**. The user must respond to the message dialog before continuing work in the current application
- **4096 System modal**. All applications are suspended until the user responds to the message dialog.

Name

Description

Changes the name of a directory or a file.

Syntax

```
Name OldName$ As NewName$
```

Parameters

OldName$ - a string expression that identifies the directory or file you wish to rename.

NewName$ - a string expression that contains the new name for the file or directory.

Notes

If you are renaming a file and it is not in the current directory you must give the full path as part of both parameters of the statement.

It is possible to use the Name procedure to move a file by giving a different path in NewName$. If you do this, the file will be moved (not copied) to the new directory.

On Error

Description

Determines the way the program reacts when an error is encountered. The first syntax causes the program to jump to the subroutine identified by label. The second syntax causes the program to resume running at the next line after the line at which the error occurred. The third syntax disables any error handling in the current procedure.

Syntax

```
On Error GoTo label
On Error Resume Next
On Error GoTo 0
```

Parameters

label - name of a label statement within the procedure.

Notes

A label statement is marked by a label name which ends with the colon " : " character. The corresponding label parameter is the label name without the colon character.

Example

The following example sets the error handling jump and then causes two errors to be generated. The error handling routing uses the Number and Description properties of the Err object to show a message describing the error. Control is then returned to the main program after the lines that generated the errors.

```
Sub Main
  On Error GoTo ErrorHandler
  Dim x as object
  x.draw        ' Object not set error
  y = 1/0     ' Division by zero
  MsgBox "You are now back in the main program."
  Exit Sub
ErrorHandler:
  Msg$ = "The following error has occurred:" & Chr(10)
  Msg$ = Msg$ & "Error number = " & Err.Number & Chr(10)
  Msg$ = Msg$ & Err.Description
  MsgBox Msg$, 16, "An error has occurred"
  Resume Next
End Sub
```

Option Base

Description

Declares the default lower bound for array subscripts.

Syntax

```
Option Base BaseNum
```

Parameters

BaseNum - can be 0 or 1.

Notes

If used, this statement can appear only once in a module, must occur only in the Declarations section, and must be used before you declare the dimensions of any arrays. If no Option Base statement is used in a module, the default base for arrays is 0.

Regardless of the setting of the Option Base, you can explicitly set the bounds for an array dimension by using a declaration of the form: Dim myArray(1 To 10).

Example

In the following example, the Option Base statement is used to set the default base for arrays to 1. Two arrays are then dimensioned: A is a ten element array using the default lower bound; B is a ten element array dimensioned using an explicit To keyword to set the lower bound to 0.

```
Option Base 1       ' Module level statement.
Sub Main
 Dim A(10)          ' Default base used
 Dim B(0 To 9)      ' To statement explicitly sets base
 MsgBox "Lower bound for A = " & LBound(A)
 MsgBox "Lower bound for B = " & LBound(B)
End Sub
```

Option Explicit

Description

Forces explicit declaration of all variables. If this statement is used, a compile error will be generated if the code contains any variables which are not explicitly declared.

Syntax

```
Option Explicit
```

Parameters

This procedure takes no parameters.

Notes

If used, this statement can appear only once in a module, must occur only in the Declarations section.

Example

Option Explicit

```
Sub Main
 y = 5 ' generates a compile error because
       ' the variable y has not been declared
End Sub
```

Print

Description

Print a string to the default message window.

Syntax

```
Print expression[, expression...]
```

Parameters

expression - any valid expression.

Notes

The default message window is a simple message box. The primary intention of the Print procedure is to provide a simple means of displaying values during program development. To display program messages to a user, use the more flexible MsgBox procedure or function.

Example

This example shows how the Print procedure can be used during program development to display information during program execution.

```
Sub PrintExample ()
 Dim Pi
 Pi = 22/7
 Print Pi    ' View the output of a calculation
End Sub
```

Rem

Description

Used to include explanatory remarks in a program.

Syntax

```
Rem remark
```

' remark

Parameters

remark - explanatory text or comment used to document a macro.

Notes

The Rem statement causes the line of text following the keyword to be treated as a comment. Comments are ignored by the interpreter. Use the single quote symbol ' to mark part of a line as a comment. All text following the ' symbol on the line is treated as a comment.

Example

The following example shows the use of comments in a procedure.

```
Sub Main()
  Rem This is a comment line
  MsgBox "hello"  ' This is a comment in a line of code
  ' This is also a comment line
End Sub
```

SendKeys

Description

Sends one or more keystrokes to the active window as if they had been entered at the keyboard.

Syntax

```
SendKeys (Keys$, [wait])
```

Parameters

Keys$ - a string expression that represents the keys to be sent to the active window.

wait - optional Boolean TRUE or FALSE. If TRUE, the keystrokes are processed before control is returned to the calling program. If wait is FALSE or omitted, control returns immediately.

Notes

The following special string sequences can be included in the Keys$ string:

Key	Code	Key	Code
ALT	%	HOME	{HOME}
CTRL	+	INS or INSERT	{INSERT} or {INS}
SHIFT	^	LEFT ARROW	{LEFT}
BACKSPACE	{BACKSPACE}	NUM LOCK	{NUMLOCK}
BREAK	{BREAK}	PAGE DOWN	{PGDN}
CAPS LOCK	{CAPSLOCK}	PAGE UP	{PGUP}
DEL or DELETE	{DELETE} or {DEL}	PRINT SCREEN	{PRTSC}
DOWN ARROW	{DOWN}	RIGHT ARROW	{RIGHT}
END	{END}	SCROLL LOCK	{SCROLLLOCK}
ENTER	{ENTER} or ~	TAB	{TAB}
ESC	{ESC}	UP ARROW	{UP}
HELP	{HELP}	F1, F2, etc	{F1}, {F2}, etc

Example

The following example opens the Windows calculator, changes the view to Scientific and multiplies two numbers together.

```
Sub Main ()
```

```
 X = Shell("Calc.exe", 1) ' Shell Calculator.
 SendKeys "%VS", True     ' Change to Scientific mode Alt-VS.
 SendKeys "12*5=", True   ' 60 should be displayed.
End Sub
```

Set

Description

Assigns an object to an object variable.

Syntax

```
Set ObjectVar = {[New] ObjectExpression | Nothing}
```

Parameters

ObjectVar - a valid object variable.

ObjectExpression - an object or object expression.

Notes

The Dim and Static statements only declare a variable that refers to an object. No actual object is referred to until you use the Set statement to assign a specific object.

The optional New keyword is usually used during declaration to enable implicit object creation. When New is used with Set, it creates a new instance of the class. If ObjectVar contained a reference to an object, that reference is released when the new one is assigned. The New keyword can't be used to create new instances of any intrinsic data type and can't be used to create dependent objects.

Using the Nothing keyword discontinues association of ObjectVar with any specific object.

Example

The following example interacts with the Visio drawing application by creating a series of objects associated with the application.

```
Sub Main
 Dim visio As Object
 Set visio = CreateObject( "visio.application" )
 Dim draw As Object
 Set draw = visio.Documents
 draw.Open "c:\visio\drawings\Sample1.vsd"
 MsgBox "Open docs: " & draw.Count
 Dim page As Object
 Set page = visio.ActivePage
 Dim red As Object
 Set red = page.DrawRectangle (1, 9, 7.5, 4.5)
 red.FillStyle = "Red fill"
 Dim cyan As Object
 Set cyan = page.DrawOval (2.5, 8.5, 5.75, 5.25)
 cyan.FillStyle = "Cyan fill"
 Dim green As Object
 Set green = page.DrawOval (1.5, 6.25, 2.5, 5.25)
 green.FillStyle = "Green fill"
 Dim DarkBlue As Object
 set DarkBlue = page.DrawOval (6, 8.75, 7, 7.75)
 DarkBlue.FillStyle = "Blue dark fill"
 visio.Quit
End Sub
```

Shell

Description

Runs an executable program file and returns the program's task ID.

Syntax

```
Shell(AppFile$ [, style])
```

Parameters

AppFile$ - a string expression that contains the name (and path) of and executable program file. Supported file types include .PIF, .COM, .BAT, or .EXE.

style - optional integer expression that sets the window style for the application when it is invoked. See below.

Notes

If the call to the application file is successful, the function returns a Long integer data type containing the task ID for the instance of the application. If the call is not successful, the function returns 0.

The way the application opens is determined by the style parameter, as follows:

style =	Window type
1,5,9	Normal with focus
2	Minimized with focus (default)
3	Maximized with focus
4,8	normal without focus
6,7	minimized without focus

Example

The following example opens the Windows calculator, changes the view to Scientific and multiplies two numbers together.

```
Sub Main ()
 X = Shell("Calc.exe", 1) ' Shell Calculator.
 SendKeys "%VS", True     ' Change to Scientific mode Alt-VS.
 SendKeys "12*5=", True   ' 60 should be displayed.
End Sub
```

Space

Description

Inserts a number of spaces in a string expression or Print# statement.

Syntax

```
Space(number )
```

Parameters

The parameter number can be any valid integer and determines the number of blank spaces.

number - an integer expression giving the number of spaces to insert.

Notes

This function gives a shorthand way of inserting a number of spaces into a string expression. The function returns a String data type which contains the designated number of spaces.

Example

The following example uses the Space function as a shorthand way of inserting a number of spaces into a string expression.

```
Sub Main
    MsgBox "Hello" & Space(20) & "There"
End Sub
```

Static

Description

Used to declare variables and allocate storage space. These variables will retain their value throughout the program run.

Syntax

```
Static VarName [As type]
```

Parameters

VarName - a valid variable name

type - a valid variable data type. Valid types are: Integer, Long, Single, Double, Date, Boolean, String, Variant, Object

Notes

Declaring a variable within a function or subprocedure with the Static statement enables the preservation of the value of the variable between calls to that function or procedure during program execution.

Example

This example defines three procedures. The main procedure calls two subprocedures a number of times. The two subprocedures are identical except that Sub2 defines a static variable whose value is retained between calls.

```
Sub Main
 For a = 1 to 4
  Sub2    ' Jump to Sub2
  Sub3    ' Jump to Sub3
 Next a
End Sub
' Define second procedure
Sub Sub2
 Static i As Integer
 i = i + 5
 MsgBox "The current value of i = " & i
End Sub
' Define third procedure
Sub Sub3
 Dim j As Integer
 j = j + 5
 MsgBox "The current value of j = " & j
End Sub
```

Stop

Description

Ends the execution of a program.

Syntax

```
Stop
```

Parameters

None

Notes

The Stop statement can be placed anywhere in your code. This statement has the same effect as the End statement.

Type...End Type

Description

Declares a user-defined data type containing one or more elements.

Syntax

```
Type usertype
```

```
   elementname As type
   [elementname As type]
   ...
End Type
```

Parameters

usertype - name of a user-defined data type. It follows standard variable naming conventions.

elementname - name of an element of the user-defined data type. It follows standard variable-naming conventions.

type - a valid data type: Integer, Long, Single, Double, String, Variant, or another user-defined type. The argument typename can't be an object type.

Notes

Once you have declared a user-defined type using the Type procedure, you can declare a variable of that type anywhere in your script. Use Dim or Static to declare a variable of a user-defined type. Line numbers and line labels aren't allowed in Type...End Type blocks.

User-defined types are often used with data records because data records frequently consist of a number of related elements of different data types. Arrays cannot be an element of a user defined type in Enable Basic.

Example

The following example creates a data type called Component which has three elements: Designator, Value and CompType. The macro dimensions a variable myComp as a Component type and asks the user to enter a value for each element of myComp.

```
Type Component              ' Create a user defined data type.
    Designator As String
    Value As Double
    CompType As String
End Type
Dim myComp As Component      ' Dimension a variable as the user type.
' ------------------------------------------------------------
' The following sub procedure asks the user for a value for each
' element, and then displays the results.
Sub Form_Click ()
    myComp.Designator = InputBox("Enter a designator(eg C5)")
    myComp.Value = CDbl(InputBox("Enter a value (number only)"))
    myComp.CompType = InputBox("Enter component type (eg Resistor)")
    Msg$ = "The component " & myComp.Designator
    Msg$ = Msg$ & " is a " & myComp.CompType
    Msg$ = Msg$ & " with a value of " & myComp.Value & " units."
    MsgBox Msg$
End Sub
```

UBound

Description

Returns the largest available subscript for the dimension of the indicated array.

Syntax

```
UBound(array [, dimension])
```

Parameters

array - a valid array name, excluding parentheses.

dimesion - a optional numeric expression representing an array dimension of a multi-dimensional array. The first dimension of an array is 1, the second 2, etc.

Notes

If dimension is omitted and array has more than one dimension, the upper bound for the first dimension is returned.

Example

The following example displays the upper and lower bounds for each dimension of a two-dimensional array.

```
Sub Main()
 Dim myArray(2 To 6, 4 To 8)
 Dim Low1, Low2, Up1, Up2, Msg
 Low1 = LBound(myArray)
 Up1 = UBound(myArray)
 Low2 = LBound(myArray, 2)
 Up2 = UBound(myArray, 2)
 MsgBox "First dimension goes from element " & Low1 & " to " & Up1
 MsgBox "The 2nd dimension goes from element " & Low2 & " to " & Up2
End Sub
```

Val

Description

Returns the numeric value of a string of characters.

Syntax

```
Val(String$)
```

Parameters

String$ - a string expression.

Notes

The function attempts to convert String$ to a number by extracting any valid numerical value contained at the beginning of the string. For example, the string "200 ohm" will return a value of 200. If the string does not begin with a numerical value, the function returns 0. The function returns a Double data type.

Example

The following example shows the return values for various strings.

```
Sub ValConvert
 Print Val("45")                ' returns 45
 Print Val("45ohm")             ' returns 45
 Print Val("a 45 ohm resistor") ' returns 0
 Print Val("45e-5 farad")       ' returns 0.00045
End Sub
```

The following example asks the user to enter a string and then displays the result of passing the string to the **Val** function.

```
Sub main
 Dim Msg
 Dim YourVal As Double
 Again = 6
 While Again = 6
  YourVal = Val(InputBox$("Enter a number or string"))
  Msg = "The number you enered is: " & YourVal
  Msg = Msg & Chr(10) & Chr(10) & "Try again?"
  Again = MsgBox(Msg, 68, "Val test")
 Wend
End Sub
```

VarType

Description

Returns a value that indicates the data type of the parameter VarName.

Syntax

```
VarType(VarName)
```

Parameters

VarName - a valid variable name.

Notes

The function returns an integer that indicates the data type according to the following table:

return value	Var Type
0	(empty) Variant
1	Null
2	Integer
3	Long
4	Single
5	Double
7	Date/Time
8	String
9	Object
11	Boolean
12	(empty) Array

Example

The following example creates variables of different types and then displays the VarType for each variable.

Sub main

```
Dim a
b = Empty
c = 5
d = 123456789
e = CSng(1.23)
f = 1.23456789
g = Date
h = Time(Now)
i = "Hello"
Dim j As Object
k = True
Dim l(3)
MsgBox "a is VarType " & VarType(a)
MsgBox "b is VarType " & VarType(b)
MsgBox "c is VarType " & VarType(c)
MsgBox "d is VarType " & VarType(d)
MsgBox "e is VarType " & VarType(e)
MsgBox "f is VarType " & VarType(f)
MsgBox "g is VarType " & VarType(g)
```

```
 MsgBox "h is VarType " & VarType(h)
 MsgBox "i is VarType " & VarType(i)
 MsgBox "j is VarType " & VarType(j)
 MsgBox "k is VarType " & VarType(k)
 MsgBox "l is VarType " & VarType(l)
End Sub
```

Enable Basic Statements

Enable Basic Operators

In an Enable Basic macro script, an operator is a character or set of characters that tells the interpreter to perform some action on one or more stated values. For example, the " * " character is used in Enable Basic to multiply two numbers together.

In this section:

Do ... Loop	End	For Next
For Each Next	Function End Function	If Then ElseIf Else
GoTo	GoSub Return	Select Case End Case
Sub End Sub	While Wend	With End With
Arithmetic operators	Comparison operators	Concatenation operators
Logical operators	Using different number bases	Operator precedence.

Do...Loop

Description

Conditional statement that repeats a group of statements while a particular condition is true, until a particular condition is met, or until an Exit Do statement is encountered.

Please note: You may include only ONE (1) instance of the [{While|Until} condition] syntax in a Do...Loop structure, i.e. it can be included at the Do end or the Loop end, but not both.

Syntax

```
Do [{While|Until} condition]
  ...statement to be executed...
[Exit Do]
  ...statement to be executed...
Loop [{While|Until} condition]
```

Parameters

condition - an expression that resolves to a TRUE or FALSE.

Notes

Putting a [{While|Until} condition] after the Do statement will cause the loop to test the condition before any statements in the loop are executed. If the condition is not true, then the loop is not executed at all. Putting a [{While|Until} condition] after the Loop statement allows the statement within the loop to execute once before the condition is tested.

Example

The following example displays an input dialog with the instructions to enter a number between 5 and 10. If the user enters a number that is not in this range, the Do...Loop re-displays the input dialog until a valid number is entered.

Sub Main

```
 Dim Value, Msg
 Do
 Value = InputBox("Enter a value from 5 to 10.")
  If Value >= 5 And Value <= 10 Then
    Exit Do          ' Exit Do...Loop.
  Else
    Beep             ' Beep if not in range.
  End If
 Loop
End Sub
```

End

Description

This procedure is used to end execution of a program or, with the optional keywords, mark the end of a function or procedure definition, or an If, With, Select Case or Type statement block.

Syntax

```
End [{Function | Sub | If | With | Select | Type}]
```

Parameters

This procedure takes no parameters

Notes

Using the End statement without any of the optional keywords in any function or procedure causesan immediate end to current program execution.

Exit

Description

Exits a loop, function or procedure.

Syntax

```
Exit {Do | For | Function | Sub }
```

Parameters

None

Example

This sample shows a Do ... Loop with an Exit Do statement to get out of the loop.

```
Sub Main ()
  Dim Value, Msg
  Do
   Value = InputBox("Enter a value from 5 to 10.")
   If Value >= 5 And Value <= 10 Then ' Check range.
      Exit Do             ' Exit Do...Loop.
   Else
      Beep                ' Beep if not in range.
   End If
  Loop
End Sub
```

For...Next

Description

Repeats the execution of a block of statements a specified number of times.

Syntax

```
For counter = StartNum% To StopNum%  [Step increment%]
  ...statements...
Next [counter]
```

Parameters

counter - a variable used to contain the loop count number.

StartNum% - an integer or integer expression used as the starting number for the loop count.

StopNum% - an integer or integer expression used as the finishing number for the loop count.

increment% - optional integer or integer expression used to define the amount the counter is incremented on each loop. If this parameter is omitted, default is 1.

Notes

The initial value of counter is set to StartNum%. When the Next statement is encountered, counter is incremented by 1 (or by increment% if this parameter is specified). If counter is less than or equal to StopNum%, the statements within the loop are executed again. Loop execution continues until counter is greater than StopNum%.

You can nest any number of For...Next loops by using different counter variables.

Example

This example shows three nested For...Next loops.

```
Sub main ()
 Dim x,y,z
 For x = 1 to 3
  For y = 1 to 3
   For z = 1 to 3
    Print "Looping" ,z,y,x
   Next z
  Next y
 Next x
End Sub
```

For Each...Next

Description

Repeats the group of statements for each element in an array of a collection.

Syntax

```
For Each element In group
  ...statements...
[Exit For]
  ...statements...
Next [element]
```

Parameters

element - a variable used to represent each element in group.

group - a variable containing an array of elements.

Notes

For Each ... Next statements can be nested if each loop element is unique. The For Each...Next statement cannot be used with an array of user defined types.

Example

This example uses the For Each procedure to step through all the elements in an array.

```
Sub Main
  dim z(1 to 4) as double
  z(1) = 1.11
  z(2) = 2.22
  z(3) = 3.33
  For Each v In z
    Print v
  Next v
End Sub
```

Function...End Function

Description

Declares and defines a procedure that can receive arguments and return a value of a specified data type.

Syntax

```
Function Fname [(ArgList)] [As type]
```

```
   ...statements...
Fname = expression
[   ...statements...
Fname = expression]
End Function
```

Parameters

Fname - the name of the function being defined.

ArgList - a list of variables to be passed to the function as arguments when the function is called. ArgList takes the form:

([ByVal] variable [As type] [, [ByVal] variable [As type]]...)

The optional ByVal keyword indicates that the argument is passed by value rather than by reference. The As keyword identifies the data type of the variable.

type - the data type of the value returned by the function.

expresssion - an expression that determines the return value of the function.

Notes

The Fname = expression statement sets the return value of the function. You must include at least one Fname = expression statement in a function to define its return value.

Example

The following example defines a function called SquareNum which returns the square of a number. The main procedure asks the user to enter a number and then passes the number to SquareNum and displays the return result.

```
Sub Main()
 Dim myInput$, mySquare, myNum
 myInput$ = InputBox("Enter Number to square.")
 If IsNumeric(myInput$) Then
  myNum = CDbl(myInput$)
  mySquare = SquareNum(myNum)                    'Call function
  MsgBox "The square of " & myNum & " is " & mySquare
 Else
  MsgBox "Sorry, you didn't enter a valid number."
 End If
End Sub
' ------------------
' Function definition for SquareNum()
' ------------------
Function SquareNum (ByVal X as Double) As Double
         SquareNum = X * X                        ' Return result
End Function
```

If...Then...ElseIf...Else

Description

Conditionally executes one or several statement blocks depending on whether condition is TRUE or FALSE.

Syntax

```
If condition Then ...statement...
If condition Then
   ...statement block...
[ElseIf condition2 Then
   ...statement block...]
[Else
   ...statement block...]
End If
```

Parameters

condition - an expression that resolves to a TRUE or FALSE value.

Notes

If there is only one statement to execute if condition is TRUE, then you may use the first described syntax, which places the entire If...Then statement on a single line. For more complex structures that include multiple execution statements or the optional ElseIf and Else clauses, use the second syntax.

The optional ElseIf clause allows another condition to be tested if the initial If condition is FALSE. You may include any number of ElseIf clauses. When using multiple ElseIf clauses only the statement block associated with the first TRUE condition statement will be executed, even if subsequent conditions are also TRUE.

The optional Else clause is executed if no previous conditions were found to be TRUE.

Examples

The following example asks the user to enter a number, date or text and then determines what data type was entered.

```
Sub Main
 Dim myInput
 myInput = InputBox("Enter a number, date or some text")
 If IsNumeric(myInput) Then
  MsgBox "You entered a number = " & CSng(myInput)
 ElseIf IsDate(myInput) Then
  MsgBox "You entered a date  = " & CDate(myInput)
 Else
  MsgBox "You entered a string = " & myInput
 End If
End Sub
```

GoTo

Description

Branches unconditionally and without return to a specified label in a procedure.

Syntax

```
GoTo label
```

Parameters

label - name of a subroutine label statement within the procedure.

Notes

This statement is an unconditional jump. There is no way to return from a Goto jump. If you wish to jump to a subroutine and return, use the GoSub procedure. A label statement is marked by a label name which ends with the colon " : " character. The corresponding label parameter is the label name without the colon character.

Example

The following example uses the GoTo procedure to jump over a subprocedure definition.

```
Sub Main
  print "hello"
  GoSub Fred          ' Jump to Fred subroutine
  Goto TheEnd         ' Jump over Fred subroutine
  Fred:               ' Definition of Fred subroutine
    print "world"
  Return              ' End of Fred
  TheEnd:
End Sub
```

GoSub...Return

Description

Branches to and returns from a subroutines defined within a function or procedure.

Syntax

```
GoSub label
...statements...
label:
...statements...
Return
```

Parameters

label - name of a subroutine label statement within the procedure.

Notes

The GoSub statement allows you to jump to points within a function or procedure definition. A subroutine label statement is marked by a label name which ends with the colon " :" character. The corresponding label parameter is the label name without the colon character.

Example

The following example defines a subprocedure called Fred within the Main procedure.

```
Sub Main
  print "hello"
  Gosub Fred
  Exit Sub
  Fred:
    print "world"
  Return
End Sub
```

Select Case...End Case

Description

Executes one of the statement blocks under a Case statement based on the value of the test variable.

Syntax

```
Select Case TestVar
Case Val1
   ...statement block...
Case Val2
   ...statement block...
Case...
   ...
[Case Else]
   ...statement block...
End Select
```

Parameters

TestVar - a valid variable whose value will be tested against each Case statement.

Val1, Val2, ... - a numeric or string expression representing the test value.

Notes

In a Select Case block, the value of the TestVar is compared to the value given in each subsequent Case statement. If the value of TestVar is equal to the test value, the statement block associated with the particular Case value is executed. The optional Case Else block is executed if the value of TestVar does not equate with any of the preceding Case values.

Example

This example determines the number entered by the user.

```
Sub Test()
```

```
 Dim myNum
 myNum = "Loop"
 While Not(IsNumeric(myNum))
  myNum = InputBox("Enter an integer number between 1 and 5")
 Wend
 Select Case myNum
  Case 1
   MsgBox "The number is one"
  Case 2
   MsgBox "The number is two"
  Case 3
   MsgBox "The number is three"
  Case 4
   MsgBox "The number is four"
  Case 5
   MsgBox "The number is five"
  Case Else
   MsgBox "The number wasn't between 1 and 5!"
 End Select
End Sub
```

Sub...End Sub

Description

Declares and defines a procedure, parameters and code.

Syntax

Sub SubName [(arglist)]

  ...Statements...

End Sub

Parameters

SubName - the name of the procedure. Follows standard Enable Basic naming conventions.

arglist - optional argument list. The list has the following syntax:

([ByVal] variable [As type][, ByVal] variable [As type] ...)

The optional ByVal parameter specifies that the variable is passed by value instead of by reference. The optional As parameter is used to specify the data type.

Notes

When calling a subprocedure from within a function or procedure, use the subprocedure name SubName. If the subprocedure takes parameters, include the necessary values or expressions after SubName with each parameter separated with a comma. Do not use parentheses to enclose the parameter list in a call to a subprocedure, unless you are using the optional Call keyword.

Example

The following example defines two procedures: one called Main and one called mySub that accepts two parameters passed by value. Main contains a call to mySub.

```
' Define the main procedure
Sub Main
   Dim a As String, b As Integer
   a = "From the main procedure"
   b = 5
   mySub a, b              ' Call mySub and pass params
   MsgBox "Back in Main"
```

```
End Sub
' Define mySub with a string and an integer parameters
Sub mySub(ByVal Param1 As String, ByVal Param2 As Integer)
   MsgBox Param1
   MsgBox "b = " & Str(Param2)
End Sub
```

While...Wend

Description

Continually loops through the statement block while condition is TRUE. The condition parameter is evaluated at the beginning of each loop.

Syntax

```
While condition
   ...Statement block...
Wend
```

Parameters

condition - any expression that evaluates to TRUE or FALSE.

Example

The following example uses a Boolean variable DoAgain to control the While loop.

```
Sub Main
 DoAgain = True
 While DoAgain
  UserInput = InputBox("Enter a number")
  DoAgain = Not(IsNumeric(UserInput))
 Wend
 MsgBox "You entered the number " & UserInput
End Sub
```

The following example uses a numeric comparison expression to control the **While** loop.

```
Sub Main
 Dim x As Integer
 x = 0
 While x < 10
  MsgBox "x is currently " & x
  x = x +2
 Wend
 MsgBox "x has reached " & x
End Sub
```

With...End With

Description

With procedure allows you to perform a series of commands or statements on a particular object without again referring to the name of that object. With statements can be nested by putting one With block within another With block. You will need to fully specify any object in an inner With block to any member of an object in an outer With block.

Syntax

```
With object
   ...statements...
End With
```

Parameters

object - a valid object, such as a user-defined variable object or dialog box object.

Notes

The With procedure is useful for simplifying coding when writing a series of statements that reference different elements or properties of a single object.

Within the With statement block you can refer to elements or properties of object directly without having to use the object name. For example, if you have defined a dialog box that contains two checkboxes with the identifiers .CheckMe1 and .CheckMe2 and then dimensioned an object variable myDlg to refer to the dialog box, you could refer to the value of the checkboxes normally using the syntax myDlg.CheckMe1 and myDlg.Checkme2. If you use the With statement of the form With myDlg , you could then refer to the values of the checkboxes using .CheckMe1 and .CheckMe2 within the With block.

Example

The following example creates two user-defined variable types and then uses the With statement block to reference the elements within each data type.

```
Type type1          ' Define a user data type
    a As Integer    ' with multiple elements.
    d As Double
    s As String
End Type
Type type2          ' Define a second user data type
    a As String
    o As type1
End Type
Dim type1a As type1 ' Declare object variables as
Dim type2a As type2 ' user data types.
' ---------------------------------------------------
Sub Main ()
    With type1a     ' Use With to reference the object.
      .a = 65       ' You can now refer to elements of the
      .d = 3.14     ' object by their identifiers.
    End With
    With type2a
      .a = "Hello, world"
      With .o       ' Nested With statement
        .s = "Goodbye"
      End With
   End With
   type1a.s = "YES"
   MsgBox type1a.a
   MsgBox type1a.d
   MsgBox type1a.s
   MsgBox type2a.a
   MsgBox type2a.o.s
End Sub
```

Arithmetic Operators in Enable Basic

The following is a list of arithmetic operators that can be used in Enable Basic, listed in the order of precedence in which they are executed:

Operator	Usage	Function

^	**x = y^z**	Exponentiation: x is y raised to the power of z
-	**x = -y**	Negation: x is the negative value of y
*	**x = y * z**	Multiplication: x is y multiplied by z
/	**x = y/z**	Division: x is y divided by z. Note: if z=0 a divide by zero error will occur.
Mod	**x = y Mod z**	Modulo: x is the remainder when y (rounded to an integer) is divided by z (rounded to an integer). Mod always returns an integer value.
+	**x = y + z**	Addition: x is y plus z.
-	**x = y - z**	Subtraction: x is y minus z

Comparison Operators in Enable Basic

The following is a list of comparison operators that can be used in Enable Basic, listed in the order of precedence in which they are executed:

Operator	Usage	Function
<	**x < y**	Less than: Returns TRUE if x is less than y, otherwise returns FALSE.
<=	**x <= y**	Less than or equal to Returns TRUE if x is less than or equal to y, otherwise returns FALSE.
=	**x = y**	Equals: Returns TRUE if x is equal to y, otherwise returns FALSE.
>	**x >= y**	Greater than or equal to: Returns TRUE if x is greater than or equal to y, otherwise returns FALSE.
>	**x > y**	Greater than: Returns TRUE if x is greater than y, otherwise returns FALSE.
<>	**x <> y**	Not equal to: Returns TRUE if x is not exactly equal to y, otherwise returns FALSE.

Comparison operators return a Boolean TRUE or FALSE which can be used as a test condition in conditional statements. Comparison operators also work with String values. In this case, the determination a "greater than" or "less than" condition is based on how the strings would compare in an alphabetical listing. String comparisons are not case sensitive, i.e. "abc" = "ABC".

Concatenation Operators in Enable Basic

The following is a list of string concatenation operators that can be used in Enable Basic, listed in the order of precedence in which they are executed:

Operator	Usage	Function
&	**x & y**	Joins the characters of string x to the characters of string y. If x or y are not string data types, they will be converted.
+	**x + y**	Joins the characters of string x to the characters of string y only if both x and y are string data types. If x or y is a number, the expression is treated as numerical and the operator acts as the numerical plus. Note: In numerical calculations, strings are converted to 0 unless they begin with a valid number, in which case they take the value of the number.

Logical Operators in Enable Basic

The following is a list of logical or Boolean operators that can be used in Enable Basic, listed in the order of precedence in which they are executed:

Operator	Usage	Function

AND	**x AND y**	Logical And: Returns TRUE if both x and y are TRUE, otherwise returns FALSE.
OR	**x Or y**	Logical Or: Returns FALSE if both x and y are FALSE, otherwise returns TRUE.

Note

x and y must be expressions that return a Boolean TRUE or FALSE value.

The **Not** function can be used to return the Boolean opposite of a logical value.

Using Different Number Bases in Enable Basic

Enable Basic supports three representations of numbers: Decimal, Octal and Hexadecimal. To use Octal (base 8) or hexadecimal (base 16) numbers in a script, prefix the number with **&O** or **&H** respectively.

Operator Precedence in Enable Basic

The following list shows the order of precedence of execution of Enable Basic operators:

Operator	Description	Order
()	parenthesis	highest
^	exponentiation	
-	negation	
/,*	division/multiplication	
Mod	modulo	
+, -	addition, subtraction	
&, +	concatenation	
=,>,<,<=,>=	relational	
AND	and	
OR	or	lowest

Enable Basic Functions

In Enable Basic, there are two main types of language elements:

- Functions – these elements perform an action and then return a value.
- Procedures – these perform actions but do not have any return value.

This section of the Enable Basic Reference details each of the functions and procedures supported by the Enable Basic scripting language:

- *Using Functions and Procedures in Enable Basic*
- *Dates and Times*
- *IO*
- *Maths*
- *Strings*
- *Enable Basic Extensions*.

Using Functions and Procedures in Enable Basic

An Enable Basic macro script must define at least one procedure - that which defines the main program code. You can, however, define other procedures and functions that can be called by your code. Procedures which are called by other procedures or functions are called subprocedures.

As with the main program code, subprocedures are defined within a **Sub...End Sub** statement block.

Functions are defined within a **Function...End Function** statement block.

Any subprocedures or functions used in a macro script must be defined after the main program procedure definition.

To use (known as "calling") a function or subprocedure, include the name of the function or procedure in a statement in the same way that you would use the built-in Enable Basic functions and procedures. If the function or procedure requires parameters, then you must include these in the calling statement.

Both functions and subprocedures can be defined to accept parameters, but only functions can be defined to return a value to the calling statement.

When calling a function, you must ensure that the statement containing the function call handles the return value. This is usually done by assigning the return value to a variable or using the return value in a calculated expression.

You may assign any name to functions and procedures that you define, as long as it conforms to the standard Enable Basic naming conventions.

In this section:

- *Defining Parametrized Enable Basic Functions and Procedures*
- *Passing Parameters to Enable Basic Functions and Procedures*
- *Calling Procedures in external DLLs from Enable Basic*.

Defining Parameterized Enable Basic Functions and Procedures

Both functions and subprocedures you define in a macro script can be declared to accept parameters. Additionally, functions are defined to return a value.

To define a parameterized function or subprocedure, list the variable names to be used to store the parameters as part of a `Sub` or `Function` statement. For example, the following code fragment defines a subprocedure that accepts two parameters, storing them in variables Param1 and Param2 respectively:

```
Sub mySub(Param1, Param2)
```

To call this subprocedure, you would use a statement of the form:

```
mySub Var1, Var2
```

where Var1 and Var2 are to be passed to the subprocedure mySub.

Optionally in a `Sub` or `Function`, you can specify the data type of the parameters by including the `As type` clause in the parameter list. For example:

```
Sub mySub(Param1 As Integer, Param2 As String)
```

A valid call to this function would be:

```
mySub 5, "Hello"
```

As well as defining the data type of the parameters to be passed, you can choose how the parameters will be passed to the subroutine or function. Enable Basic provides two methods of passing parameters to subprocedures or functions:

- `ByRef` - passes a reference to the variable passed and allows the subprocedure or function to make changes to the actual variables that are passed in as parameters. This is the default method of passing parameters and is used if the method is not explicitly declared.
- `ByVal` - passes the value of the variable only. The subprocedure or function can use this value, but the original variable passed is not altered.

For more information on using the `ByRef` and `ByVal` keywords, see the following section, Passing Parameters to Enable Basic Functions and Procedures.

Passing Parameters to Enable Basic functions and procedures

When you define a function or subprocedure in a macro script that can accept parameters, you can pass variables to the function or subprocedure in two ways: by reference or by value. To declare the method that parameters are passed, use the `ByRef` or `ByVal` keywords in the parameter list when defining the function or subprocedure in a `Sub` or `Function` statement. For example, the following code fragment defines a subprocedure that accepts two parameters. The first is passed by value and the second by reference:

```
Sub Test (ByVal Param1 As Integer , ByRef B As String)
```

The difference between the two methods is that `ByRef` passes a reference to the variable passed and allows the subprocedure or function to make changes to the actual variables that are passed in as parameters (this is the default method of passing parameters and is used if the method is not explicitly declared). `ByVal` passes the value of the variable only. The subprocedure or function can use this value, but the original variable passed is not altered.

The following examples illustrate the differences between methods. The main procedure is as follows:

```
Sub Main
 Dim X As Integer, Y As String
 X = 45 : Y = "Number"
 Test X, Y      ' Call to a subprocedure called Test.
 Print X, Y
End Sub
```

The above procedure includes a call to a subprocedure, Test. If the subprocedure is defined as follows:

```
Sub Test (ByRef A As Integer , ByRef B As String)
 B = B & " = " & A : A = 10*A
End Sub
```

then the variables X and Y in the main procedure are referenced directly by the subprocedure. The result is that the values of X and Y are altered by the subprocedure so that after the Test is executed X = 450 and Y = "Number = 45".

If, however, the subprocedure is defined as follows:

```
Sub Test (ByVal A As Integer , ByVal B As String)
 B = B & " = " & A : A = 10*A
End Sub
```

then after Test is executed X = 45 and Y = "Number", i.e. they remain unchanged.

If the subprocedure is defines as follows:

```
Sub Test (ByRef A As Integer , ByVal B As String)
 B = B & " = " & A : A = 10*A
End Sub
```

then after Test is executed, X = 450 and Y = "Number". Because Y was passed by value, it remains unchanged.

You can override the `ByRef` setting of a function or subprocedure by putting parentheses around a variable name in the calling statement. Calling Test with the following statement:

```
Test (X), Y
```

would pass the variable X by value, regardless of the method defined for that parameter in the procedure definition.

Calling Procedures in external DLLs from Enable Basic

You can call a function or procedure which is in an external dynamic linked library (DLL) from an Enable Basic macro script by first defining the function or procedure using the `Declare` statement. The `Declare` statement must by placed outside the main procedure in the Global Declarations section of the file. All declarations of external functions and procedures are Global and accessible by all procedures and functions. If the call to the DLL returns a value, declare it as a function, otherwise declare it as a subprocedure.

The following code segment declares a function called GetPrivateProfileString which resides in `Kernel32.dll`, which is part of the Windows system. The function takes six parameters and returns an integer value.

```
Declare Function GetPrivateProfileString Lib "Kernel32" _
(ByVal lpApplicationName As String, ByVal lpKeyName As _
String, ByVal lpDefault As String, ByVal lpReturnedString _
As String, ByVal nSize As Integer, ByVal lpFileName As String)_
As Integer
```

The following code segment declares a subprocedure called InvertRect which is in `User.dll`, also part of the Windows system. The procedure takes two parameters. Being a subprocedure, it does not return any values.

```
Declare Sub InvertRect Lib "User" (ByVal hDC As Integer, _ aRect As Rectangle)
```

Once an external function or procedure is declared, you can call it just as you would any other internal or user-defined function or procedure.

It is important to note that the Enable Basic interpreter cannot verify that you have declared an external DLL function or procedure using the correct name or parameters. Nor can it determine if you have passed correct values to the external DLL. Before declaring and using an external function or subprocedure, it is important that you understand the workings of the external function/procedure.

Dates and Times

Date

Description

The function returns a Date data type containing the current date in the system short date format.

Syntax

```
Date
```

Parameters

No parameters.

Example

This example displays the current date in two formats.

```
Sub Main
 Dim Msg, NL, d1, d2
 NL = Chr(10)        ' Define newline.
 d1 = Date
 d2 = Format (Date, "d-mmmm-yy")
 Msg = "Date 1 is " & d1 & NL
 Msg = Msg & "Date 2 is " & d2 & NL
 MsgBox Msg
End Sub
```

Day

Description

The function returns an integer between 1 and 365, inclusive that represents the day of the month given in DateVar$. To check that a string or string expression can be resolved to a valid date, use the IsDate function.

Syntax

```
Day(DateVar$)
```

Parameters

DateVar$ - a string or string expression that can be resolved to a valid date.

Example

The following example displays the current day, month and year is succession.

```
Sub Main
 MsgBox "The current day of the month is " & Day(Date)
 MsgBox "The current month is " & Month(Date)
 MsgBox "The current year is " & Year(Date)
End Sub
```

Hour

Description

The function returns an integer between 0 and 23 that is the hour of the day indicated in the parameter TimeVar$. TimeVar$ must be able to be resolved into a valid time, for example "08:04:23 PM" or "20:04:23". The function returns an Integer data type.

Syntax

```
Hour(TimeVar$)
```

Parameters

TimeVar$ - a string expression that represents a valid time.

Example

This example displays the hours, minutes and seconds derived from a time string.

```
Sub Main
 MyTime = "08:04:23 PM"
 MsgBox Second( MyTime ) & " Seconds" ' returns 23
 MsgBox Minute( MyTime ) & " Minutes" ' returns 4
 MsgBox Hour( MyTime ) & " Hours"     ' returns 8
End Sub
```

IsDate

Description

Returns a value that indicates whether a variant parameter can be converted to a date. The function returns a Boolean TRUE if expression can be resolved to a valid date, or FALSE if it cannot.

Syntax

```
IsDate(expression)
```

Parameters

An expression to be validated.

Minute

Description

The function returns an integer between 0 and 59 representing the minute of the hour given in TimeVar$.TimeVar$ must be able to be resolved into a valid time, for example "08:04:23 PM" or "20:04:23".

Syntax

```
Minute(TimeVar$)
```

Parameters

TimeVar$ - a string expression that represents a valid time.

Example

This example displays the hours, minutes and seconds derived from a time string.

```
Sub Main
 MyTime = "08:04:23 PM"
```

```
 MsgBox Second( MyTime ) & " Seconds" ' returns 23
 MsgBox Minute( MyTime ) & " Minutes" ' returns 4
 MsgBox Hour( MyTime ) & " Hours"     ' returns 8
End Sub
```

Month

Description

The function returns an integer between 1 and 12, inclusiverepresenting the month of the year given in DateVar$. To check that a string or string expression can be resolved to a valid date, use the IsDate function.

Syntax

```
Month(DateVar$)
```

Parameters

DateVar$ - a string expression that can be resolved to a valid date.

Example

The following example displays the current day, month and year is succession.

```
Sub Main
 MsgBox "The current day of the month is " & Day(Date)
 MsgBox "The current month is " & Month(Date)
 MsgBox "The current year is " & Year(Date)
End Sub
```

Now

Description

This function returns a Date data type which includes both the current date and time values.

Syntax

```
Now
```

Parameters

None

Second

Description

Returns an integer between 0 and 59 representing the seconds portion of the time given in TimeVar$. TimeVar$ must be able to be resolved into a valid time, for example "08:04:23 PM" or "20:04:23".

Syntax

```
Second(TimeVar$)
```

Parameters

TimeVar$ - a string or string expression that represents a valid time.

Example

This example displays the hours, minutes and seconds derived from a time string.

```
Sub Main
 MyTime = "08:04:23 PM"
 MsgBox Second( MyTime ) & " Seconds" ' returns 23
 MsgBox Minute( MyTime ) & " Minutes" ' returns 4
 MsgBox Hour( MyTime ) & " Hours"     ' returns 8
End Sub
```

Time

Description

The function returns the current system time in the default system short time format.

Syntax

Time

Parameters

None

Example

The following example gets the current system time and formats it for display in the Long Time format.

```
Sub Main
' Returns current system time in the long time format.
  MsgBox Format(Time, "Long Time")
End Sub
```

Year

Description

The function returns an integer that represents the year given in DateVar$. To check that a string or string expression can be resolved to a valid date, use the IsDate function.

Syntax

```
Year(DateVar$)
```

Parameters

DateVar$ - a string expression that can be resolved to a valid date.

Example

The following example displays the current day, month and year is succession.

```
Sub Main
 MsgBox "The current day of the month is " & Day(Date)
 MsgBox "The current month is " & Month(Date)
 MsgBox "The current year is " & Year(Date)
End Sub
```

IO

ChDir

Description

Changes the current default directory.

Syntax

```
ChDir PathName$
```

Parameters

PathName$ - a string or string expression, limited to fewer then 128 characters, identifying the path to target directory. If the drive letter is omitted from Pathname$, the function operates on the current drive.

Notes

To return the current default directory, use the CurDir and CurDrive functions.

Example

This example changes to the root directory and then back to the original directory.

```
Sub Main ()
 Dim Answer, Msg, NL       ' Declare variables.
 NL = Chr(10)              ' Define newline.
 CurPath = CurDir()        ' Get current path.
 ChDir "\"                 ' Change to root directory
 Msg = "The current directory has been changed to "
 Msg = Msg & CurDir() & NL & NL & "Change back."
 Answer = MsgBox(Msg)      ' Get user response.
```

```
 ChDir CurPath              ' Change back to user default.
 Msg = "Directory changed back to " & CurPath & "."
 MsgBox Msg                 ' Display results.
End Sub
```

ChDrive

Description

Changes the current default drive.

Syntax

```
ChDrive Drivename$
```

Parameters

DriveName$ - a string or string expression that references a valid drive.

Notes

Only a single letter is used by the ChDrive procedure to identify a target drive. If more than a single character is supplied as an argument, the procedure ignores all but the first character. Therefore the statements ChDrive("D:") and ChDrive("D") are equivalent. Both will change to the D: drive, however the colon in the first statement is ignored and is not necessary.

Example

This example changes to G: drive and back.

```
Sub Main ()
 Dim Msg, NL        ' Declare variables.
 NL = Chr(10)       ' Define newline.
 CurPath = CurDir() ' Get current path.
 ChDrive "G"       ' Change to drive G:
 Msg = "The current directory has been changed to "
 Msg = Msg & CurDir() & NL & NL & "Change back."
 MsgBox Msg         ' Get user response.
 ChDir CurPath      ' Change back to user default.
 Msg = "Directory changed back to " & CurPath & "."
 MsgBox Msg ' Display results.
End Sub
```

Chr

Description

Converts an integer representing an ASCII character code into the corresponding character.

Syntax

```
Chr(int)
```

Parameters

int - an integer expression which represents an ASCII character code.

Notes

The function returns a string containing a single character whose character code is the int argument. To convert a single character string to an ASCII code, use the Asc function.

Example

This example prints all the letters of the alphabet into a message dialog.

```
Sub ChrExample ()
 Dim X, Msg
 For X = Asc("A") To Asc("Z")
  Msg = Msg & Chr(X)
 Next X
```

```
 MsgBox Msg
End Sub
```

Close

Description

Closes all active input/output files.

Syntax

```
Close
```

Parameters

This procedure takes no parameters

Notes

Whenever you open a file with the Open procedure, ensure you issue a Close statement after performing all file operations. This ensures the files are properly released by the program.

Example

This example creates three files, writes some text to these files and the closes the files.

```
Sub Make3Files ()
 Dim I, FNum, Fname    ' Declare variables.
 For I = 1 To 3
  FNum = FreeFile      ' Determine next file number.
  FName = "TEST" & FNum
  Open FName For Output As Fnum  ' Open file.
  Print #I, "This is test #" & I ' Write string.
  Print #I, "Here is another ";  "line";  I
 Next I
 Close                ' Close all files.
End Sub
```

CurDir

Description

Returns the full path including drive name to the current directory.

Syntax

```
CurDir
```

Parameters

This function takes no parameters.

Notes

The function returns a String data type. The function is read only. To change the current drive and directory use the ChDrive and ChDir functions.

Example

The following example displays the current directory in a message box.

```
Sub Form_Click ()
 Dim Msg, NL        ' Declare variables.
 NL = Chr(10)       ' Define newline.
 Msg = "The current directory is: "
 Msg = Msg & NL & CurDir
 MsgBox Msg         ' Display message.
End Sub
```

EOF

Description

Returns a value (0 = false, -1 = true) during file input that indicates whether the end of the file has been reached.

Syntax

```
EOF(FileNumber)
```

Parameters

FileNumber - an integer that is the file number of an open input file.

Notes

The function returns a -1 value if the end of the file has been reached, otherwise it returns 0.

Example

This example uses the Input function to read 10 characters at a time from a file and display them in a message dialog. This example assumes that TESTFILE is a text file with a few lines of sample data.

```
Sub Main
    Open "TESTFILE" For Input As #1
    Do While Not EOF(1)         ' Loop until end of file.
      MyStr = Input(10, #1)     ' Get ten characters.
      MsgBox MyStr
    Loop
    Close #1                    ' Close file.
End Sub
```

Erase

Description

Reinitializes the elements of a fixed array.

Syntax

```
Erase arrayname
```

Parameters

arrayname - the name of an array variable. Specify the array name only, without any subscripts or parentheses.

Notes

Reinitializing an array sets all elements in the array to 0 for numeric arrays, or a null string for string arrays.

Example

This example defines two arrays, one integer and one string and assigns values to them. The Erase procedure is then used to reinitialize the arrays.

```
Sub Main
    Dim IntArray(2) As Integer, StrArray(2) As String
    Dim I As Integer
    ' Initialize the arrays
    For I = 0 to 2
        IntArray(I) = I+1
    Next I
    StrArray(0) = "Goodbye"
    StrArray(1) = "cruel"
    StrArray(2) = "world!"
    ' Display the arrays
    Print IntArray(0), IntArray(1), IntArray(2)
    Print StrArray(0), StrArray(1), StrArray(2)
    ' Reinitialize the arrays
    Erase IntArray
    Erase StrArray
    ' Redisplay the arrays
```

```
    Print IntArray(0), IntArray(1), IntArray(2)
    Print StrArray(0), StrArray(1), StrArray(2)
End Sub
```

FileLen

Description

Returns a long integer that is the length of the file in bytes. This function returns a Long data type. If the file does not exist, the function returns –1.

Syntax

```
FileLen(FileName$)
```

Parameters

FileName$ - a string or string expression that contains the filename and path of the file to be tested.

Example

The following example displays the size of the config.sys file.

```
Sub Main()
 Dim myFile$, Size&
 myFile$ = "C:\config.sys"
 Size& = FileLen(myFile$)
 If Size& = -1 Then        ' Test that file was found
   MsgBox "Could not find file."
 Else
   MsgBox "Config file is " & Size& & " bytes"
 End If
End Sub
```

FileCopy

Description

Copies a file from source to destination.

Syntax

```
FileCopy SourceFile$, DestinationFile$
```

Parameters

SourceFile$ - a string expression containing the file name, including full path if not in the current directory, of the file to copy.

DestinationFile$ - a string expression containing the path and file name for the copied file.

Notes

For both SourceFile$ and DestinationFile$ you must specify a name for the file, even if the destination file is to be of the same name as the original. If the source or destination is not the current directory, you must specify the path as part of the SourceFile$ and/or DestinationFile$ strings.

Note: You must independently determine if the source file exists as the FileCopy function does not test for this. If SourceFile$ does not exist, the statement will create a zero length file of the name and at the location given by DestinationFile$. If DestinationFile$ already exists, it will NOT be overwritten. If the path specified in DestinationFile$ does not exist, it will NOT be created.

Example

The following example copies the file test.txt from the temp1 directory on C: drive to the temp2 directory. The copied file is given the name newname.txt. This example assumes C:\temp1\test.txt exists and that the directory C:\temp2 has been created.

```
Sub ExpExample ()
 Dim mySource, myDestination As String
 mySource = "C:\temp1\test.txt"
 myDestination = "C:\temp2\newname.txt"
 FileCopy mySource, MyDestination
```

```
End Sub
```

Input

Description

Input returns a defined number of characters from a sequential file.

Syntax

```
Input(numBytes , [#]FileNumber)
```

Parameters

numBytes - numeric expression representing the number of bytes to be read from the file.

FileNumber - the file number used in the Open statement to open the file.

Notes

The file referenced by FileNumber must be opened for input using an Open statement before it can be used by the Input function. The function returns a String data type.

Example

This example assumes that a file called TESTFILE exists in the current directory.

```
Sub Main
  Open "TESTFILE" For Input As #1    ' Open file.
  Do While Not EOF(1)                ' Loop until end of file.
   MyStr = Input(10, #1)         ' Get ten characters.
   MsgBox MyStr
  Loop
  Close #1                            ' Close file.
End Sub
```

Kill

Description

This procedure is used to delete a file from a disk.

Syntax

```
Kill FileName$
```

Parameters

FileName$ - a string expression that contains the full filename and path of a file to delete.

Notes

This procedure is only used to remove files. To remove a directory, use the RmDir procedure.

Example

The following example creates three test files in the current directory and then deletes them.

```
Const NumberOfFiles = 3
Sub Main ()
 Dim Msg, J
 Msg = "Creating test files in the "
 Msg = Msg & CurDir & " directory."
 Msg = Msg & Chr(10) & "Files will be deleted after use."
 If MsgBox(Msg, 1, "Test") = 1 Then
    Call MakeFiles()          ' Create data files.
    Msg = "Three test files have been created on your disk "
    Msg = Msg & "in " & CurDir & Chr(10)
    Msg = Msg & "Choose OK to remove these files."
    If MsgBox(Msg, 1, "Remove files?") = 1 Then
       For J = 1 To NumberOfFiles
```

```
          Kill "TEST" & J       ' Remove data files from disk.
       Next J
    MsgBox "Files have been removed."
    End If
 End If
End Sub
' Function to generate test files
Sub MakeFiles ()
 Dim I, FNum, FName
 For I = 1 To NumberOfFiles
     FNum = FreeFile       ' Determine next file number.
     FName = "TEST" & I
     Open FName For Output As FNum      ' Open file.
     Print #FNum, "This is test #" & I
     Print #FNum, "Here is another ";  "line";  I
 Next I
 Close                     ' Close all files.
End Sub
```

Line Input

Description

Reads a line from a sequential file into a String or Variant variable.

Syntax

```
Line Input #FileNumber, VarName
```

Parameters

The parameter FileNumber is used in the Open statement to open the file. The parameter VarName is the name of a variable used to hold the line of text from the file.

FileNumber - the number used in an Open statement to open the file.

VarName - the name of a string or variant variable to hold the line of text.

Notes

The file referred to by FileNumber must be opened for input using an Open statement prior to using this function.

Example

This example assumes a file called TESTFILE exists in the current directory.

```
Sub Main
 Open "TESTFILE" For Input As #1    ' Open file.
 Do While Not EOF(1)                ' Loop until end of file.
  Line Input #1, TextLine           ' Read line into variable.
  Print TextLine                    ' Print to Debug window.
 Loop
 Close #1                          ' Close file.
End Sub
```

MkDir

Description

Creates a new directory within an existing directory.

Syntax

```
MkDir Path$
```

Parameters

Path$ - a string expression of less than 128 characters that represents the path and name of the directory to create.

Notes

You can only use `MkDir` to create a new directory within a directory that already exists. You cannot create multiple directories in a hierarchy with a single `MkDir` statement. For example, if the directory `C:\TEST` does not exist, then the statement `MkDir "C:\TEST\SUBTEST"` would not perform any action. You would first need to create `C:\TEST` and then create `C:\TEST\SUBTEST`.

If Path$ already exists, the procedure performs no action.

Example

This example creates a directory on C: drive called NewDir1 and then creates a subdirectory under this called NewDir2. The macro then removes the newly created directories.

```
Sub Main
   Dim DST As String
   DST = "C:\NewDir1"
   If MsgBox("OK to create test directories?", 1, "New Dir") = 1 Then
    MkDir DST
    MsgBox "New directory has been created on C:"
    MkDir "C:\NewDir1\NewDir2"
    MsgBox "A new subdirectory has been created."
    If MsgBox("OK to delete test directories?", 1, "Del Dir") = 1 Then
     RmDir "C:\NewDir1\NewDir2"
     RmDir DST
     MsgBox "Test directories removed."
    End If
   End If
End Sub
```

Open

Description

Opens a file for sequential input and output operations.

Syntax

```
Open FileName$ [For mode] [Access access] As [#]FileNumber
```

Parameters

FileName$ - a string expression that contains the filename and path.

mode - keyword that specifies the file mode: Append, Input, Output

access - keyword that specifies which operations are permitted on the file: Read, Write, Read Write.

FileNumber - integer expression with a value between 1 and 255, inclusive. When an Open statement is executed, filenumber is associated with the file as long as it is open. Other I/O statements can use the number to refer to the file.

Notes

You must open a file before any I/O operation can be performed on it. If FileName$ doesn't exist, it is created when a file is opened for Append or Output modes. If the file is already opened by another process and the specified type of access is not allowed, the Open operation fails and a permission denied error occurs.

mode =

Input Sequential input mode. You can read data from the file.

Output Sequential output mode. You can write data to the file.

Append Append sets the file pointer to the end of the file. A Print # or Write # statement then adds to (appends to) the file.

access =

Read - Opens the file for reading only.

Write - Opens the file for writing only.

Read Write - Opens the file for both reading and writing. This mode is valid only for files opened for Append mode.

Example

The following example creates a file called TESTFILE in the current directory and then gets input from the user and writes this to the file. The program then reads the data in the file and displays it.

```
Sub Main ()
' Create a test file and write some date to it
 Open "TESTFILE" For Output As #1 ' Open to write file.
 userData1$ = InputBox ("Enter your own text here")
 userData2$ = InputBox ("Enter more of your own text here")
 Write #1, "This is a test file."
 Write #1, userData1$
 Write #1, userData2$
 Close #1
' Read the data back from the test file
 Open "TESTFILE" For Input As #2 ' Open to read file.
 Msg$ = ""
 Do While Not EOF(2)
  Line Input #2, FileData ' Read a line of data.
  Msg$ = Msg$ & FileData & Chr(10) ' Construct message.
 Loop
 Close #2  ' Close all open files.
 MsgBox Msg$ ' Display file contents
 Kill "TESTFILE" ' Remove file from disk.
End Sub
```

Print

Description

Writes data to a sequential file.

Syntax

```
Print #FileNumber, [OutputList]
```

Parameters

FileNumber - the filenumber associated with the file to print to.

OutputList - list of expressions to print (see below)

The OutputList argument takes the following format:

[{Spc(n) | Tab[(n)]}] [expression] [charpos]

where:

Spc(n) - Used to insert space characters in the output, where n is the number of space characters to insert.

Tab(n) - Used to position the insertion point to an absolute column number, where n is the column number. Use Tab with no argument to position the insertion point at the beginning of the next print zone.

Expression - Numeric expressions or string expressions to print.

charpos - Specifies the insertion point for the next character. Use a semicolon to position the insertion point immediately after the last character displayed. Use Tab(n) to position the insertion point to an absolute column number. Use Tab with no argument to position the insertion point at the beginning of the next print zone. If charpos is omitted, the next character is printed on the next line.

Notes

If you omit OutputList and include only a list separator after FileNumber, a blank line is printed to the file. Multiple expressions can be separated with either a space or a semicolon. A space has the same effect as a semicolon.

The Print # statement usually writes a Variant data type to a file the same way it writes any other data type. However, there are some exceptions:

If the data being written is a Variant of VarType 0 (Empty), Print # writes nothing to the file for that data item.

If the data being written is a Variant of VarType 1 (Null), Print # writes the literal #NULL# to the file.

If the data being written is a Variant of VarType 7 (Date), Print # writes the date to the file using the system Short Date format. When either the date or the time component is missing or zero, Print # writes only the part provided to the file.

Example

The following example prints lines of date to a sequential file:

```
Sub Main
 Dim I, FNum, FName ' Declare variables.
 For I = 1 To 3
  FNum = FreeFile ' Determine next file number.
  FName = "TEST" & FNum
  Open FName For Output As FNum ' Open file.
  Print #I, "This is test #" & I ' Write string to file.
  Print #I, "Here is another ";  "line";  I
 Next I
 Close ' Close all files.
End Sub
```

RmDir

Description

Removes an existing empty directory.

Syntax

```
RmDir DirName$
```

Parameters

DirName$ - a string expression that contains the name and path to the directory to be removed.

Notes

If the directory listed in DirName$ does not exist, the procedure takes no action. If the directory referred to in DirName$ contains files, then no action is taken. To remove a directory it must first be empty.

Example

This example creates a directory on C: drive called NewDir1 and then creates a subdirectory under this called NewDir2. The macro then removes the newly created directories.

```
Sub Main
   Dim DST As String
   DST = "C:\NewDir1"
   If MsgBox("OK to create test directories?", 1, "New Dir") = 1 Then
    MkDir DST
    MsgBox "New directory has been created on C:"
    MkDir "C:\NewDir1\NewDir2"
    MsgBox "A new subdirectory has been created."
    If MsgBox("OK to delete test directories?", 1, "Del Dir") = 1 Then
     RmDir "C:\NewDir1\NewDir2"
     RmDir DST
     MsgBox "Test directories removed."
    End If
   End If
End Sub
```

Seek

Description

The function form of Seek returns the byte position of the file pointer in an open file. This marks the current read/write position in the file.

The procedure form of Seek sets the read/write position of an open file.

Syntax

```
Seek([#]FileNumber)
Seek [#]FileNumber, Position
```

Parameters

FileNumber - a numeric expression that represents a number used in the Open # statement.

Position - a numeric expression representing the byte position relative to the beginning of the file.

Notes

The function form of Seek returns a Long data type. The FileNumber parameter must reference a currently open file.

Example

The following examples assume that a file called TESTFILE containing several lines of data exists in the current directory. This example uses the function form of Seek to print the current byte position.

```
Sub SeekFunction
  Open "TESTFILE" For Input As #1 ' Open file for reading.
  Do While Not EOF(1)             ' Loop until end of file.
   MyChar = Input(1, #1)          ' Read next character of data.
   Print Seek(1)                  ' Print byte position.
  Loop
  Close #1                        ' Close file.
End Sub
```

This example uses the procedure form of **Seek** to move through the data.

```
Sub SeekProcedure
  Open "TESTFILE" For Input As #1 ' Open file for reading.
  For i = 1 To 24 Step 3          ' Loop until end of file.
   Seek #1, I                     ' Seek to byte position
   MyChar = Input(1, #1)          ' Read next character of data.
   Print MyChar                   ' Print character of data.
  Next i
  Close #1                        ' Close file.
End Sub
```

Write

Description

Writes and formats data to a sequential file that has been opened in Output or Append mode.

Syntax

```
Write #FileNumber [, paramrlist ]
```

Parameters

FileNumber - a valid file number that references a file opened with the Open # statement.

paramlist - a comma-delimited list of numeric or string expressions to write to the file.

Notes

A comma delimited list of the supplied parameters is written to the indicated file. If no parameters are present, the newline character is all that will be written to the file.

Example

The following example creates a file called TESTFILE in the current directory and then gets input from the user and writes this to the file. The program then reads the data in the file and displays it.

```
Sub Main ()
' Create a test file and write some date to it
 Open "TESTFILE" For Output As #1 ' Open to write file.
```

```
 userData1$ = InputBox ("Enter your own text here")
 userData2$ = InputBox ("Enter more of your own text here")
 Write #1, "This is a test file."
 Write #1, userData1$
 Write #1, userData2$
 Close #1
' Read the data back from the test file
 Open "TESTFILE" For Input As #2 ' Open to read file.
 Msg$ = ""
 Do While Not EOF(2)
  Line Input #2, FileData ' Read a line of data.
  Msg$ = Msg$ & FileData & Chr(10) ' Construct message.
 Loop
 Close #2  ' Close all open files.
 MsgBox Msg$ ' Display file contents
 Kill "TESTFILE" ' Remove file from disk.
End Sub
```

Maths

Cos

Description

Returns the cosine of an angle. The argument must be expressed in radians and must be a valid numeric expression. The function returns a Double data type.

Syntax

```
Cos (number)
```

Parameters

number - a numeric variable, constant or expression.

Example

The example displays the cosine of 1, 2, 3 radians.

```
Sub Main()
 Dim I As Single            'Declare variables.
 For I =1 To 3
  Msg = Msg & "Cos(" & I & ")="
  Msg = Msg & Cos(I) & " " 'Cos function call
 Next I
 MsgBox Msg                 ' Display results.
End Sub
```

Exp

Description

Returns the base of the natural log raised to a power (e ^ num).

Syntax

```
Exp(num)
```

Parameters

num - number or numeric expression.

Notes

The value of the constant e is approximately 2.71828. This function returns a Double data type.

Example

This example calculates the value of the constant e.

```
Sub ExpExample ()
  ' Exp(x) is e ^x so Exp(1) is e ^1 or e.
  Dim Msg, ValueOfE      ' Declare variables.
  ValueOfE = Exp(1)      ' Calculate value of e.
  Msg = "The value of e is " & ValueOfE
  MsgBox Msg
End Sub
```

Log

Description

Returns the natural log of a number.

Syntax

```
Log(num)
```

Parameters

num - a numerical expression. The number must greater than zero.

Notes

The base for the natural log is the constant e, which is approximately 2.71828. This function returns a Double data type.

Not

Description

Returns the Boolean opposite of expression.

Syntax

```
Not(expression)
```

Parameters

expression - an expression that resolves to a Boolean TRUE or FALSE.

Notes

The expression parameter must be able to be resolved to a TRUE or FALSE value. A numeric 0 value is the equivalent of a Boolean FALSE. Any non-zero numeric value resolves to a Boolean TRUE.

Oct

Description

Returns the octal value of the decimal parameter. The function returns a String data type which is the octal representation of the argument.

Syntax

```
Oct(num)
```

Parameters

num - a numeric expression.

Rnd

Description

Returns a random number between 0 and 1. The function returns a Double data type whose value is randomized between 0 and 1.

Syntax

```
Rnd
```

Parameters

None

Example

The example uses the Rnd function to simulate rolling a pair of dice by generating random values from 1 to 6.

```
Sub Main ()
 Dim Dice1, Dice2, Msg, Again
 Again = True
 While Again
  MsgBox "Press OK to roll the dice"
  Dice1 = CInt(5 * Rnd) + 1    ' Generate first die value.
  Dice2 = CInt(5 * Rnd) + 1    ' Generate second die value.
  Msg = "You rolled a " & Dice1
  Msg = Msg & " and a " & Dice2
  Msg = Msg & " for a total of "
  Msg = Msg & Str(Dice1 + Dice2) & "." & Chr(10)
  Msg = Msg & Chr(10) & "Roll again?"
  If MsgBox(Msg, 36, "Roll dice") = 7 Then
   Again = False
  End If
 Wend
End Sub
```

Sin

Description

Returns the sine of an angle. The argument must be expressed in radians and must be a valid numeric expression. The function returns a Double data type.

Syntax

```
Sin(number)
```

Parameters

number - a numeric variable, constant or expression.

Example

```
Sub Main ()
 pi = 4 * Atn(1)
 rad = 90 * (pi/180)
 x = Sin(rad)
 Print x
End Sub
```

Sqr

Description

Returns the square root of a number.

Syntax

```
Sqr(num)
```

Parameters

The parameter num must be a valid number greater than or equal to zero.

num - a positive number or numeric expression.

Notes

The function returns a Double data type representing the square root of num.

Tan

Description

Returns the tangent of an angle.

Syntax

```
Tan(number)
```

Parameters

number - a numeric expression.

Notes

The argument must be expressed in radians and must be a valid numeric expression. The function returns a Double data type.

Example

The following example calculates the tangent of Pi/4 radians.

```
Sub Main
    Dim Msg, Pi
    Pi = 4 * Atn(1)      ' Calculate Pi.
    Msg = "Pi is equal to " & Pi
    MsgBox Msg
    x = Tan(Pi/4)
    MsgBox  x & " is the tangent of Pi/4"
End Sub
```

Strings

LCase

Description

Returns a string in which all letters of the string$ parameter have been converted to lower case.

Syntax

```
LCase(string$)
```

Parameters

string$ - a string or string expression.

Notes

The function returns a String data type.

Len

Description

Returns the number of characters in a string.

Syntax

```
Len(string$)
```

Parameters

string$ - a string expression.

Notes

The function returns an Integer data type representing the length of the string.

Left

Description

Returns the left most number of characters of a string parameter.

Syntax

```
Left(string$, chars)
```

Parameters

string$ - a string expression.

chars - the number of characters to return.

Notes

The function returns a String data type. If chars is greater than the number of characters in the string, then the whole string is returned.

Example

This example shows the use of the Left and Right string functions.

```
Sub Main ()
 myString = "Hello out there"
 MsgBox Left(myString, 5)     ' Returns "Hello"
 MsgBox Mid(myString, 7, 3)   ' Returns "out"
 MsgBox Right(myString, 5)    ' Returns "there"
End Sub
```

Mid

Description

Returns a substring within a string.

Syntax

```
Mid(String$, StartPos, NumChar)
```

Parameters

String$ - a string expression.

StartPos - a numeric expression that gives the start position within Str$ for the return substring. StartPos is the number of characters from the start of Str$.

NumChar - a numeric expression that represents the number of characters to return.

Notes

The function returns a String data type. If StartPos is greater than the length of String$, a null string is returned. If NumChar extends beyond the end of String$, all characters from StartPos to the end of String$ are returned.

Example

This example shows the use of the Left and Right string functions.

```
Sub Main ()
 myString = "Hello out there"
 MsgBox Left(myString, 5)     ' Returns "Hello"
 MsgBox Mid(myString, 7, 3)   ' Returns "out"
 MsgBox Right(myString, 5)    ' Returns "there"
End Sub
```

Right

Description

Returns a number of characters from the end of the string parameter.

Syntax

```
Right(string$, chars )
```

Parameters

string$ - a string or string expression.

chars - an integer expression defining the number of characters to return.

Notes

The function returns a String data type which is the right-most chars characters of string$. If chars is greater than the length of string$, then the return value is string$.

Example

This example shows the use of the Left and Right string functions.

```
Sub Main ()
 myString = "Hello out there"
 MsgBox Left(myString, 5)     ' Returns "Hello"
 MsgBox Mid(myString, 7, 3)   ' Returns "out"
```

```
 MsgBox Right(myString, 5)    ' Returns "there"
End Sub
```

Str

Description

Returns the value of a numeric expression as a string.

Syntax

```
Str(number)
```

Parameters

number - a number or numeric expression.

Notes

The function returns a String data type.

Example

The following example determines the position of a particular digit in a number by converting the number to a string and then using the InStr search function.

```
Sub main ()
 Dim a As Long
 Dim x As Integer
 Dim aStr As String
 a = 16374859
 aStr = Str(a)            ' Convert a to a string.
 x = InStr(1, aStr, "7")  ' Search for a particular digit.
 MsgBox "7 is digit number " & x & " in " & aStr
End Sub
```

Trim, LTrim, RTrim

Description

LTrim, RTrim and Trim all return a copy of a string with leading, trailing or both leading and trailing spaces removed.

Syntax

```
[L | R]Trim (String$ )
```

Parameters

String$ - a string or string expression.

Notes

Ltrim removes leading spaces. Rtrim removes trailing spaces. Trim removes leading and trailing spaces. All functions return a String data type.

Example

This example uses the LTrim and RTrim functions to strip leading and trailing spaces, respectively, from a string variable. It also uses the Trim function alone to strip both types of spaces.

```
Sub Main
  MyString = "    <-Trim->    "  ' Initialize string.
  TrimString = LTrim(MyString)
  MsgBox "|" & TrimString & "|"
  TrimString = RTrim(MyString)
  MsgBox "|" & TrimString & "|"
  TrimString = LTrim(RTrim(MyString))
  MsgBox "|" & TrimString & "|"
' Using the Trim function alone achieves the same result
' as the previous statement.
  TrimString = Trim(MyString)
```

```
  MsgBox "|" & TrimString & "|"
End Sub
```

UCase

Description

Returns a string in which all letters of the string$ parameter have been converted to upper case.

Syntax

```
UCase(string$)
```

Parameters

string$ - a string expression.

Notes

The function returns a String data type.

Enable Basic Extensions

AltKeyDown

Description

Returns a value that indicates the state of the ALT key.

Syntax

```
AltKeyDown
```

Parameters

This function takes no parameters.

Notes

The function returns 1 if the ALT key is down, otherwise it returns 0. Use the CBool function to convert the return value to a TRUE or FALSE.

ConfirmNoYes

Description

Displays a message dialog with a YES button and NO button. The title of the message box is "Confirm".

Syntax

```
ConfirmNoYes Message$, IntVar
```

Parameters

Message$ - a string expression to be used as the displayed message.

IntVar - a valid integer variable name which is used to store the value of the key pressed.

Notes

When the user exists the message box, the variable IntVar is set to 1 if the YES button was clicked, or 0 if the NO button was clicked.

The variable IntVar must be declared as an Integer data type before calling the function.

Example

The following example displays an Altium Designer confirmation dialog. If the user clicks the YES button, a message is displayed.

```
Sub Main
 Dim myVal As Integer
 ConfirmNoYes "Do you want to display a message?", myVal
 If myVal = 1 Then            ' If the user presses YES
  MsgBox "You pressed the YES button"
 End If
End Sub
```

ConfirmNoYesCancel

Description

Displays a message dialog with a YES button, NO button, CANCEL button and question mark icon.

Syntax

```
ConfirmNoYesCancel Message$, IntVar
```

Parameters

Message$ - a string expression to be used as the displayed message.

IntVar - a valid integer variable name which is used to store the value of the key pressed.

Notes

When the user exits the message box, the variable IntVar is set to 6 if the YES button was clicked, 7 if the NO button was clicked, or 2 if the CANCEL button was clicked.

The variable IntVar must be declared as an Integer data type before calling the function.

Example

The following example displays an Altium Designer confirmation dialog and displays a message if the YES or NO buttons are pressed.

```
Sub Main
 Dim myVal As Integer
 ConfirmNoYesCancel "Click YES or NO to display a message?", myVal
 Select Case myVal
  Case 6   ' YES button pressed
   MsgBox "You pressed the YES button"
  Case 7   ' NO button pressed
   MsgBox "You pressed the NO button"
 End Select
End Sub
```

GetCurrentWindowHandle

Description

Returns an integer representing the system ID handle of the currently active window.

Syntax

```
GetCurrentWindowHandle IntVar
```

Parameters

IntVar a previously declared variable of Integer data type in which to store the ID handle.

Notes

Whenever the Windows operating system creates a display window, it creates a unique ID number for the window. This extension allows you to determine the Windows ID number for the active window.

The number returned by this extension is not used directly by any other Enable Basic function, procedure or extension. It may be useful, however, if you are declaring functions within the Windows system DLLs for use with your macro.

ResetCursor

Description

Changes the cursor to the default arrow cursor.

Syntax

```
ResetCursor
```

Parameters

This extension takes no parameters.

RunApplication

Description

Runs an application program.

Syntax

```
RunApplication AppName$
```

Parameters

AppName$ - a string expression containing the name and path of an executable program file.

Notes

This extension uses the RunApplication process to run an executable file. This extension has the same function as the Shell procedure.

Example

This example runs the Notepad application.

```
Sub Main
 RunApplication "NOTEPAD.EXE"
End Sub
```

SetCursorBusy

Description

Changes the cursor to the default busy cursor.

Syntax

```
SetCursorBusy
```

Parameters

This extension takes no parameters

Notes

Use ResetCursor to reset the cursor to its default arrow icon.

ShiftKeyDown

Description

Returns a value that indicates the state of the SHIFT key.

Syntax

```
ShiftKeyDown
```

Parameters

None

Notes

The function returns 1 if the SHIFT key is down, otherwise it returns 0. Use the CBool function to convert the return value to a TRUE or FALSE.

ShowError

Description

Displays a Warning dialog containing an OK button and the warning icon.

Syntax

```
ShowError Message$
```

Parameters

Message$ - a string expression containing the message to display in the warning box.

Notes

This extension displays a warning message. It does not cause an error to be generated.

This extension is identical to the ShowWarning extension.

ShowInfo

Description

Displays an information dialog containing an OK button and the information icon.

Syntax

```
ShowInfo Message$
```

Parameters

Message$ - a string expression containing the message to display in the information box.

ShowWarning

Description

Displays a warning dialog containing an OK button and the warning icon.

Syntax

```
ShowWarning Message$
```

Parameters

Message$ - a string expression containing the message to display in the warning box.

Notes

This extension is identical to the ShowError extension.

Enable Basic Forms and Components

In this section:

Creating Custom Dialogs in Enable Basic	*Begin Dialog End*	*CancelButton*
Checkbox	*ComboBox*	*Dialog*
DlgEnable	*DlgText*	*DlgVisible*
DropListBox	*GroupBox*	*ListBox*
OKButton	*OptionButton*	*OptionGroup*
PushButton	*Text*	*TextBox*

Creating Custom Dialogs in Enable Basic

Enable Basic includes support for the use of custom dialogs within a script. A dialog is essentially a window that includes various controls that a user can manipulate. In Enable Basic, you define a dialog by including control definition statements in a `Begin Dialog...End Dialog` statement block. The following code fragment shows a simple dialog definition that just includes an **OK** button, a **Cancel** button and a single checkbox:

```
Begin Dialog ButtonSample 162,129, 180, 96, "Simple dialog"
  OKButton 84,76,40,14
  CancelButton 132,76,40,14
  CheckBox 64,28,48,16, "Click me", .CheckBox_1
End Dialog
```

In the above code, the `Begin Dialog` statement includes the name of dialog, ButtonSample, and the text that will appear on the title bar of the dialog, "Simple dialog".

To display a dialog that has been defined using the `Begin Dialog...End Dialog` statement block, you must first declare an object variable in which to store the various dialog control values using the `Dim` statement. The following code fragment dimensions a variable, myDlg, to hold the values of the ButtonSample dialog defined above:

```
Dim myDlg As ButtonSample
```

After the dialog is displayed and the user clicks on a button to exit the dialog, the values of the dialog controls (in this case `.CheckBox_1`) will be stored in the variable myDlg.

To display a dialog, use the `Dialog` function. This function displays the dialog associated with the given object variable and returns a number that corresponds to the button that the user clicks to close the dialog. The following code fragment displays the dialog declared above:

```
WhichButton = Dialog(myDlg)
```

where WhichButton is a variable used to store the return value.

Once the dialog has been displayed and closed by the user, you can interrogate the object variable to determine the values of the dialog controls when the dialog was closed. For example, the following code fragment displays the value of the checkbox once the dialog is closed.

```
MsgBox myDlg.CheckBox_1
```

Note how the checkbox control value is referenced. Any control that returns a value can be referenced using the syntax:

```
DlgVar.CtrlIdentifier
```

where DlgVar is the object variable associated with the dialog and CtrlIdentifier is the control identifier set in the control definition statement.

Note

Every custom dialog box must contain at least one control button: **OKButton**, **CancelButton** or **PushButton**.

Linking a Function to a Dialog in Enable Basic

Enable Basic gives you the ability to link a function to a user-defined dialog. The linked function is then called when the dialog is displayed and each time the user performs an action in the dialog, such as changing the value of a control. The ability to link a function to a dialog makes it possible to create nested dialogs and to process messages from a dialog without closing it.

A dialog-linked function is called:

- When the dialog is first initialized
- Each time the user changes the value of or uses a dialog control.

To link a function to a dialog, include the name of the function, preceded by the period "**.**" character, in the `Begin Dialog` statement. The following code fragment shows a dialog definition statement that links the function myDlgFunction to the dialog:

```
Begin Dialog UserDialog1 60,60, 260, 188, "A dialog", .myDlgFunction
```

Once you have linked a function to a dialog in a `Begin Dialog` statement, you must then define the function using a `Function...End Function` statement block. Each time the linked function is called, three parameters are passed to it. The first is a string containing the name of the control that was changed or activated to invoke the function call. The second is an integer value that indicates whether the function is being called during the dialog initialization (a value of 1), or whether the function call is a response to action taken in the dialog (a value of 2). The third parameter is an integer that supplies supplemental information about the control that initiated the function call. For example, if a checkbox is changed, the supplemental value represents the state of the checkbox: 1 for checked, 0 for not checked

The dialog linked function must therefore be declared using the following syntax:

```
Function FunctionName(ControlID$, Action%, SuppValue%) As Integer
```

The function returns an integer value indicating whether the dialog should remain open after the function has executed. A return value of 0 (the default) indicates that the dialog should close. Any other value indicates the dialog should remain open.

Dialog-Linked Function example

```
Sub Main
 Begin Dialog UserDialog1 60,60, 165, 120, "MyDlg 1", .DlgFunction
  GroupBox 8,45,150,50, "This is a group box:", .GroupBox1
  CheckBox 8,21,148,16, "Check to display more controls", .Chk1
  PushButton 30,66,100,16, "Open Second Dlg", .SecDlgBut
  OKButton 84,104,26,12
  CancelButton 124,104,26,12
 End Dialog
 Dim Dlg1 As UserDialog1
 If Dialog( Dlg1 ) = -1 Then
  MsgBox "You pressed the OK button"
 Else
  MsgBox "You pressed the Cancel button"
 End If
End Sub
' Define the linked function
```

```
Function DlgFunction( ControlID$, Action%, SuppValue%) As Integer
 Begin Dialog USERDIALOG2 150,150, 155, 100, "MyDlg 2", .DlgFunction
  Text 8,6,101,16, "This is a secondary dialog"
  PushButton 35,23,80,16, "Display message", .PushBut1
  CancelButton 112,84,26,12
  CheckBox 8,48,136,20, "Keep dialog open after button press", .CheckBoxKeepOpen
 End Dialog
 Dim Dlg2 As UserDialog2
 Select Case Action%
  Case 1                ' Executed when dialog initializes
   DlgEnable "GroupBox1", 0
   DlgVisible "SecDlgBut", 0
  Case 2                ' Executed each time a control is changed
   If ControlID$ = "Chk1" Then
    DlgEnable "GroupBox1"
    DlgVisible "SecDlgBut"
   ElseIf ControlID$ = "SecDlgBut" Then
    DlgFunction = 1
    x = Dialog( Dlg2 )
   ElseIf ControlID$ = "PushBut1" Then
    If Dlg2.CheckBoxKeepOpen = 1 Then
     DlgFunction = 1    ' Keeps dialog open
    Else
     DlgFunction = 0    ' Closes dialog
    End If
    MsgBox "You pushed me!"
   Else
    DlgFunction = 0
   End If
 End Select
End Function
```

Syntax of a Dialog-Linked Function in Enable Basic

Syntax

```
Function FnName(ControlID$, Action%, SuppVal%) As Integer
...statement block...
[FunctionName = ReturnValue%]
End Function
```

Parameters

FnName	-	Name of the function. Must be the same as that defined in the **Begin Dialog** statement.
ControlID$	-	String containing the control identifier for the control that initiated the function call.
Action%	-	Integer. Is 1 if the function is being called during dialog initialization. Is 2 if the function is being called in response to a user action in the dialog.
SuppVal%	-	Integer. The value depends on the type of control (See below)
ReturnVal%	-	Integer representing the return value of the function. A dialog-linked function returns a value when the user chooses a command button. Enable Basic acts on the value returned. If the value is 0 (zero), the calling dialog is closed when the function has executed. If a non-zero

		value is returned, the dialog remains open.

By returning a non-zero value from the function you can keep the dialog open. This allows you to execute more than one command from the same dialog.

Values for SuppVal%

Control	Values for SuppVal%
CheckBox	0 if cleared, 1 if checked.
OptionButton	Number of the option button selected, where 0 (zero) is the first option button within a group.
OKButton	1
CancelButton	2
other controls	Not applicable.

Begin Dialog ... End Dialog

Description

Encloses the definition of a user-created dialog box. Statements enclosed by the Begin Dialog ... End Dialog structure define the controls that appear in the dialog.

Syntax

```
Begin Dialog DlgName, PosX, PosY, Width, Height, dlgTitle$ [, .dlgFtnIdentifier]
...
dialog box controls definition procedures
...
End Dialog
```

Parameters

This procedure accepts no parameters.

DlgName - the object variable used to reference the dialog within the program. DlgName must follow standard function naming conventions.

PosX, PosY - a number or numerical expression representing the position of the top-left corner of the dialog. The position is measured in pixels from the top-left corner of the Altium Designer window.

Width - a number or numerical expression representing the width of the dialog in pixels.

Height - a number or numerical expression representing the height of the dialog in pixels

dlgTitle$ - a string or string expression representing the text to be displayed in the title bar of the dialog.

.dlgFtnIdentifier - optional identifier which defines a function to be executed whenever a change is made to the dialog by the user. .dlgFunctionIdentifier is the name of a valid function preceded by the period character.

Notes

If the optional .dlgFtnIdentifier parameter is used, you must define a function of the same name (less the initial period character) that will be executed whenever the dialog is opened or any control within the dialog is changed by the user. This dialog function is typically used to test and act upon the state of the dialog controls before the user closes the dialog. For example, the dialog function could enable or disable certain dialog controls when a particular option is selected.

Example

This example creates a dialog called DialogName1 with the title of "ASC - Hello".

```
Sub Main ()
 ' Define the dialog box.
 Begin Dialog DialogName1 60, 60, 160, 70, "ASC - Hello"
  TEXT 10, 10, 28, 12, "Name:"
  TEXTBOX 42, 10, 108, 12, .nameStr
  TEXTBOX 42, 24, 108, 12, .descStr
  CHECKBOX 42, 38, 48, 12, "&CHECKME", .checkInt
```

```
  OKBUTTON 42, 54, 40, 12
 End Dialog
 ' Dimension an object to represent the dialog.
 Dim Dlg1 As DialogName1
 Dlg1.checkInt = 1            ' Check the checkbox.
 Dialog Dlg1                  ' Display the dialog.
End Sub
```

CancelButton

Description

Used in a dialog definition for placing a CANCEL button in a user-defined dialog box.

Syntax

```
CancelButton PosX, PosY, Width, Height
```

Parameters

PosX,

Pos Y - numbers or numeric expressions representing the position of the top-left corner of the button expressed in pixels relative to the top-left corner of the dialog box.

Width - a number or numeric expression representing the width of the button in pixels.

Height - a number or numeric expression representing the height of the button in pixels.

Notes

This procedure is only used as part of a dialog definition.

CheckBox

Description

This procedure is used as part of a dialog box definition to create a check box object.

Syntax

```
CheckBox PositionX, PositionY, Width, Height, Caption$, .Identifier
```

Parameters

PosX, PosY - a numerical expression defining the x and y position of the checkbox in pixels relative to the top right of the dialog box.

Width - a numerical expression defining the width of the checkbox in pixels. This width includes the title text area.

Height - a numerical expression defining the height of the checkbox in pixels. This height includes the title text area.

Caption$ - string expression that defines the text associated with the check box.

.Identifier - name used as the identifier for the checkbox control. The name must start with the period character.

Notes

The .Identifier parameter is used to set or test the value of the check box. If the check box is checked, the .Indetifier parameter returns a value of 1. If it is not checked, it returns a value of 0.

Example

This example creates a checkbox in a dialog called "ASC - Hello". Before the dialog is displayed, the checkbox identifier is set to 1 indicating that the box is checked.

```
Sub Main ()
 ' Define the dialog box.
 Begin Dialog DialogName1 60, 60, 160, 70, "ASC - Hello"
  TEXT 10, 10, 28, 12, "Name:"
  TEXTBOX 42, 10, 108, 12, .nameStr
  TEXTBOX 42, 24, 108, 12, .descStr
  CHECKBOX 42, 38, 48, 12, "&CHECKME", .checkInt
  OKBUTTON 42, 54, 40, 12
```

```
 End Dialog
 ' Dimension an object to represent the dialog.
 Dim Dlg1 As DialogName1
 Dlg1.checkInt = 1   ' Check the checkbox.
 Dialog Dlg1    ' Display the dialog.
End Sub
```

ComboBox

Description

Used in a dialog definition for placing a Combo box in a user-defined dialog box.

Syntax

```
ComboBox PosX, PosY, Width, Height, ArrayVar(), .Identifier
```

Parameters

PosX,Pos Y - numbers or numeric expressions representing the position of the top-left corner of the box expressed in pixels relative to the top-left corner of the dialog box.

Width - a number or numeric expression representing the width of the box in pixels.

Height - a number or numeric expression representing the height of the box in pixels.

ArrayVar() - a valid array used as the source for the combo box value list.

.Identifier - name used as the identifier for the control. The name must start with the period character.

Notes

A combo box is a combination of a text entry box and a list box. The control presents a list of options for the user to choose from, but also allows the user to enter text directly.

The array named in the parameter ArrayVar() is the source of the combo box list.

The .Identifier parameter is used to reference the text value of control. If myDlg is the object variable declared to contain the dialog box values, the text value of the combo box is contained in myDlg.Identifier.

Example

You can use a list box, drop-down list box, or combo box to present a list of items from which the user can select. A drop-down list box saves space (it can drop down to cover other dialog box controls temporarily). A combo box allows the user either to select an item from the list or type in a new item. The items displayed in a list box, drop-down list box, or combo box are stored in an array that is defined before the instructions that define the dialog.

```
Sub Main
 Dim MyList$ (5)
 MyList (0) = "line Item 1"
 MyList (1) = "line Item 2"
 MyList (2) = "line Item 3"
 MyList (3) = "line Item 4"
 MyList (4) = "line Item 5"
 MyList (5) = "line Item 6"
 Begin Dialog BoxSample 16, 35, 256, 89, "Dialog Example"
  OKButton 204, 24, 40, 14
  CancelButton 204, 44, 40, 14
  ListBox 12, 24, 48, 40, MyList$( ), .Lstbox
  DropListBox 124, 24, 72, 40, MyList$( ), .DrpList
  ComboBox 68, 24, 48, 40, MyList$( ), .CmboBox
  Text 12, 12, 32, 8, "List Box:"
  Text 124, 12, 68, 8, "Drop-Down List Box:"
  Text 68, 12, 44, 8, "Combo Box:"
 End Dialog
 Dim Dlg1 As BoxSample
```

```
 Dlg1.CmboBox = MyList$(0)
 Button = Dialog(Dlg1)
 If Button = -1 Then
  Msg$ = "List Box = " & MyList$(Dlg1.Lstbox) & Chr(10)
  Msg$ = Msg$ & "Drop List Box = " & MyList$(Dlg1.DrpList) & Chr(10)
  Msg$ = Msg$ & "Combo Box = " & Dlg1.CmboBox & Chr(10)
  MsgBox Msg$
 End If
End Sub
```

Dialog

Description

The first listed syntax displays the dialog defined by DialogRecord. The function version (the second listed syntax) displays the dialog defined by DialogRecord and returns a number corresponding to the pushbutton the user presses.

Syntax

```
Dialog DialogRecord
Var = Dialog(DialogRecord)
```

Parameters

DialogRecord - the name of the dialog to display. DialogRecord must be defined in a preceding Dim statement.

Var - a variable to hold the returned button value.

Notes

Use the Begin Dialog ... End Dialog syntax to define the dialog box. Use the Dim procedure to dimension an object to represent the dialog. The variable used in the Dim procedure is the calling name of the dialog, DialogRecord.

The function form of Dialog returns an Integer data type which represents the control button pressed to close the dialog:

Returned value for button pressed:

OK button -1

CANCEL button 0

Other command button???The first PushButton defined in the dialog definition returns the number 1, the second the number 2, etc.

Example

The following example defines a dialog with a text input box and three buttons: an OK button, a CANCEL button and a general control pushbutton. The procedure displays the dialog and, when the user presses a button, displays the text typed into the text input box and which control button was pressed.

```
Sub Main ()
 Dim WhichButton%
 ' Define the dialog box.
 Begin Dialog DialogName1 58,60, 161, 65, "Press a button"
  Text 2,2,64,12, "Type your name:"
  TextBox 6,14,148,12, .nameStr
  Text 16,32,124,12, "And then press a button..."
  OKButton 16,48,40,12
  CancelButton 60,48,36,12
  PushButton 100,48,40,12, "Push me", .PushButton_1
 End Dialog
 ' End of dialog definition
 Dim Dlg1 As DialogName1
 WhichButton% = Dialog(Dlg1)    ' Display the dialog and return
                                ' the button pressed
 MsgBox "Your name is " & Dlg1.nameStr _
```

```
        & " and you pressed button " & WhichButton%
End Sub
```

DlgEnable

Description

This procedure is used in a dialog-linked function to enable or disable a particular control on a dialog box.

Syntax

```
DlgEnable CtrlName$[, Value]
```

Parameters

CtrlName$ - string or string expression that is the name of a control on a dialog box.

Value - optional integer used to determine whether the control is enabled or disabled. If Value is 1 then the control is enabled. If Value is 0 then the control is disabled. If Value is omitted then the status of the control is toggled.

Notes

The CtrlName$ parameter is derived from the control identifier used in the dialog definition statements. CtrlName$ is the control's identifier less the initial period character. For example, if a dialog definition contains the check box definition statement:

```
CheckBox 8, 46, 100, 12, "A CheckBox", .checkbox1
```

then the control identifier is .checkbox1 and the CtrlName$ parameter used to identify the control in the DlgEnable statement would be "checkbox1".

When a control is disabled, it is "grayed" out in the dialog and unavailable for user input.

DlgText

Description

This procedure is used in a dialog-linked function to set or change the text label of a dialog control.

Syntax

```
DlgText CtrlName$, Value$
```

Parameters

CtrlName$ - a string expression that is the name of a control on a dialog box.

Value$ - a string expression that contains the text to assign to the control label

Notes

The CtrlName$ parameter is derived from the control identifier used in the dialog definition statements. CtrlName$ is the control's identifier less the initial period character. For example, if a dialog definition contains the check box definition statement:

```
CheckBox 8, 46, 100, 12, "A CheckBox", .checkbox1
```

then the control identifier is .checkbox1 and the CtrlName$ parameter used to identify the control in the DlgText statement would be "checkbox1".

This statement only affects the text label associated with a control. To set the text value of a TextBox control, use a variable assignment statement of the form:

```
myDlg.TextBox1 = "Default text"
```

DlgVisible

Description

This procedure is used in a dialog-linked function to hide or make visible a particular control on a dialog box.

Syntax

```
DlgVisible CtrlName$[, Value]
```

Parameters

CtrlName$ - string or string expression that is the name of a control on a dialog box.

Value - optional integer used to determine whether the control is visible or hidden. If Value is 1 then the control is visible. If Value is 0 then the control is hidden. If Value is omitted then the status of the control is toggled.

Notes

The CtrlName$ parameter is derived from the control identifier used in the dialog definition statements. CtrlName$ is the control's identifier less the initial period character. For example, if a dialog definition contains the check box definition statement:

```
CheckBox 8, 46, 100, 12, "A CheckBox", .checkbox1
```

then the control identifier is .checkbox1 and the CtrlName$ parameter used to identify the control in the DlgVisible statement would be "checkbox1".

When a control is hidden it is not displayed in the dialog and is not accessible for user input.

DropListBox

Description

Used in a dialog definition for placing a drop-down list box in a user-defined dialog box.

Syntax

```
DropListBox PosX, PosY, Width, Height, ArrayVar(), .Identifier
```

Parameters

PosX,

Pos Y - numeric expressions representing the position of the top-left corner of the box expressed in pixels relative to the top-left corner of the dialog box.

Width - numeric expression representing the width of the box in pixels.

Height - numeric expression representing the height of the box in pixels.

ArrayVar() - a valid array used as the source for the box value list.

.Identifier - name used as the identifier for the control. The name must start with the period character.

Notes

A drop list box has a data display box with an arrow button at the side. Clicking the arrow expands the box to show a list of values that the user can choose from. The selected item is displayed in the display box. Unlike a Combo box, only values from the list can be chosen. The user cannot directly enter text into a drop list box.

The array named in the parameter ArrayVar() is the source of the displayed list. The default value displayed is the first array item.

The .Identifier parameter is used to reference the value of control. If myDlg is the object variable declared to contain the dialog box values, the value of the drop list box is contained in myDlg.Identifier.

The value of a drop list box is the index value of the array element chosen, NOT the element itself.

Example

You can use a list box, drop-down list box, or combo box to present a list of items from which the user can select. A drop-down list box saves space (it can drop down to cover other dialog box controls temporarily). A combo box allows the user either to select an item from the list or type in a new item. The items displayed in a list box, drop-down list box, or combo box are stored in an array that is defined before the instructions that define the dialog.

```
Sub Main
 Dim MyList$ (5)
 MyList (0) = "line Item 1"
 MyList (1) = "line Item 2"
 MyList (2) = "line Item 3"
 MyList (3) = "line Item 4"
 MyList (4) = "line Item 5"
 MyList (5) = "line Item 6"
 Begin Dialog BoxSample 16, 35, 256, 89, "Dialog Example"
  OKButton 204, 24, 40, 14
  CancelButton 204, 44, 40, 14
  ListBox 12, 24, 48, 40, MyList$( ), .Lstbox
  DropListBox 124, 24, 72, 40, MyList$( ), .DrpList
  ComboBox 68, 24, 48, 40, MyList$( ), .CmboBox
  Text 12, 12, 32, 8, "List Box:"
```

```
  Text 124, 12, 68, 8, "Drop-Down List Box:"
  Text 68, 12, 44, 8, "Combo Box:"
 End Dialog
 Dim Dlg1 As BoxSample
 Dlg1.CmboBox = MyList$(0)
 Button = Dialog(Dlg1)
 If Button = -1 Then
  Msg$ = "List Box = " & MyList$(Dlg1.Lstbox) & Chr(10)
  Msg$ = Msg$ & "Drop List Box = " & MyList$(Dlg1.DrpList) & Chr(10)
  Msg$ = Msg$ & "Combo Box = " & Dlg1.CmboBox & Chr(10)
  MsgBox Msg$
 End If
End Sub
```

GroupBox

Description

Used in a dialog definition for placing a box around a group of controls in a user-defined dialog box.

Syntax

```
GroupBox PosX, PosY, Width, Height, Caption$[, .Identifier]
```

Parameters

PosX,

PosY - numeric expressions representing the position of the top-left corner of the box expressed in pixels relative to the top-left corner of the dialog box.

Width - numeric expression representing the width of the box in pixels.

Height - a numeric expression representing the height of the box in pixels.

Caption$ - a string expression containing the title to appear on the box.

.Identifier - optional name used as the identifier for the control. The name must start with the period character.

Notes

This procedure is only used as part of a dialog definition. The group box has no inherent value and cannot be altered by the user.

The use of the .Identifier parameter is optional. You may want to include this if you wish to, for example, enable or disable the group box in a dialog-linked function.

Example

You can use option buttons to allow the user to choose one option from several options. Typically, you would use a group box to surround a group of option buttons, but you can also use a group box to set off a group of check boxes or any related group of controls.

```
Sub Main
 Begin Dialog GroupSample 31, 32, 185, 96, "Groups Example"
  OKButton 28, 68, 40, 14
  CancelButton 120, 68, 40, 14
  GroupBox 12, 8, 72, 52, "GroupBox", .GroupBox1
  GroupBox 100, 12, 72, 48, "GroupBox", .GroupBox2
  OptionGroup .OptionGroup1
  OptionButton 16, 24, 54, 8, "0ption 1"
  OptionButton 16, 40, 54, 8, "0ption 2"
  CheckBox 108, 24, 45, 8, "CheckBox 1", .CheckBox1
  CheckBox 108, 40, 45, 8, "CheckBox 2", .CheckBox2
 End Dialog
```

```
 Dim Dlg1 As GroupSample
 If Dialog ( Dlg1 ) Then
  Select Case Dlg1.OptionGroup1
   Case 0
    MsgBox "Option 1 was selected"
   Case 1
    MsgBox "Option 2 was selected"
  End Select
 End If
End Sub
```

ListBox

Description

Used in a dialog definition for placing a list box in a user-defined dialog box.

Syntax

```
ListBox PosX, PosY, Width, Height, ArrayVar(), .Identifier
```

Parameters

PosX,

Pos Y - numeric expressions representing the position of the top-left corner of the box expressed in pixels relative to the top-left corner of the dialog box.

Width - a numeric expression representing the width of the box in pixels.

Height - a numeric expression representing the height of the box in pixels.

ArrayVar() - a valid array used as the source for the box value list.

.Identifier - name used as the identifier for the control. The name must start with the period character.

Notes

A list box presents the user with a list of items from which to choose. The selected item is highlighted. The user cannot directly enter text into a list box.

The array named in the parameter ArrayVar() is the source of the displayed list. The default value highlighted is the first array item.

The .Identifier parameter is used to reference the value of control. If myDlg is the object variable declared to contain the dialog box values, the value of the list box is contained in myDlg.Identifier.

Please note: The value of a list box is the index value of the array element chosen, NOT the element itself.

Example

You can use a list box, drop-down list box, or combo box to present a list of items from which the user can select. A drop-down list box saves space (it can drop down to cover other dialog box controls temporarily). A combo box allows the user either to select an item from the list or type in a new item. The items displayed in a list box, drop-down list box, or combo box are stored in an array that is defined before the instructions that define the dialog.

```
Sub Main
 Dim MyList$ (5)
 MyList (0) = "line Item 1"
 MyList (1) = "line Item 2"
 MyList (2) = "line Item 3"
 MyList (3) = "line Item 4"
 MyList (4) = "line Item 5"
 MyList (5) = "line Item 6"
 Begin Dialog BoxSample 16, 35, 256, 89, "Dialog Example"
  OKButton 204, 24, 40, 14
  CancelButton 204, 44, 40, 14
  ListBox 12, 24, 48, 40, MyList$( ), .Lstbox
```

```
  DropListBox 124, 24, 72, 40, MyList$( ), .DrpList
  ComboBox 68, 24, 48, 40, MyList$( ), .CmboBox
  Text 12, 12, 32, 8, "List Box:"
  Text 124, 12, 68, 8, "Drop-Down List Box:"
  Text 68, 12, 44, 8, "Combo Box:"
 End Dialog
 Dim Dlg1 As BoxSample
 Dlg1.CmboBox = MyList$(0)
 Button = Dialog(Dlg1)
 If Button = -1 Then
  Msg$ = "List Box = " & MyList$(Dlg1.Lstbox) & Chr(10)
  Msg$ = Msg$ & "Drop List Box = " & MyList$(Dlg1.DrpList) & Chr(10)
  Msg$ = Msg$ & "Combo Box = " & Dlg1.CmboBox & Chr(10)
  MsgBox Msg$
 End If
End Sub
```

OKButton

Description

Used in a dialog definition for placing an OK button in a user-defined dialog box.

Syntax

```
OKButton PosX, PosY, Width, Height
```

Parameters

PosX,

Pos Y - numeric expressions representing the position of the top-left corner of the button expressed in pixels relative to the top-left corner of the dialog box.

Width - a numeric expression representing the width of the button in pixels.

Height - a numeric expression representing the height of the button in pixels.

Notes

This procedure is only used as part of a dialog definition.

Example

This example creates a dialog called DialogName1 with the title of "ASC - Hello".

```
Sub Main ()
 ' Define the dialog box.
 Begin Dialog DialogName1 60, 60, 160, 70, "ASC - Hello"
  TEXT 10, 10, 28, 12, "Name:"
  TEXTBOX 42, 10, 108, 12, .nameStr
  TEXTBOX 42, 24, 108, 12, .descStr
  CHECKBOX 42, 38, 48, 12, "&CHECKME", .checkInt
  OKBUTTON 42, 54, 40, 12
 End Dialog
 ' Dimension an object to represent the dialog.
 Dim Dlg1 As DialogName1
 Dlg1.checkInt = 1            ' Check the checkbox.
 Dialog Dlg1                  ' Display the dialog.
End Sub
```

OptionButton

Description

This procedure is used as part of a dialog box definition to create an option or radio button object.

Syntax

```
OptionButton PosX, PosY, Width, Height, Caption$[, .Identifier]
```

Parameters

PosX, PosY - a numerical expression defining the x and y position of the control in pixels relative to the top right of the dialog box.

Width - a numerical expression defining the width of the control in pixels. This width includes the title text area.

Height - a numerical expression defining the height of the control in pixels. This height includes the title text area.

Caption$ - string expression that defines the text label associated with the control.

.Identifier - name used as the identifier for the control. The name must start with the period character.

Notes

Option buttons are usually used in groups to present a series of options from which the user can pick only one.

A group of OptionButton statements in a dialog definition must be preceded by an OptionGroup statement, which assigns the group that the option buttons belong to. To determine which option in a group is checked, you query the option group identifier, which returns the number of the option button which is checked. The first option button defined after the OptionGroup statement is button 0, the second is button 1, etc. Checking one option button unchecks all other option buttons in the same group.

The .Identifier parameter is optional, but should be included if you wish to refer to the control in a dialog-linked function.

Example

You can use option buttons to allow the user to choose one option from several options. Typically, you would use a group box to surround a group of option buttons, but you can also use a group box to set off a group of check boxes or any related group of controls.

```
Sub Main
 Begin Dialog GroupSample 31, 32, 185, 96, "Groups Example"
  OKButton 28, 68, 40, 14
  CancelButton 120, 68, 40, 14
  GroupBox 12, 8, 72, 52, "GroupBox", .GroupBox1
  GroupBox 100, 12, 72, 48, "GroupBox", .GroupBox2
  OptionGroup .OptionGroup1
  OptionButton 16, 24, 54, 8, "0ption 1"
  OptionButton 16, 40, 54, 8, "0ption 2"
  CheckBox 108, 24, 45, 8, "CheckBox 1", .CheckBox1
  CheckBox 108, 40, 45, 8, "CheckBox 2", .CheckBox2
 End Dialog
 Dim Dlg1 As GroupSample
 If Dialog ( Dlg1 ) Then
  Select Case Dlg1.OptionGroup1
   Case 0
    MsgBox "Option 1 was selected"
   Case 1
    MsgBox "Option 2 was selected"
  End Select
 End If
End Sub
```

OptionGroup

Description

Precedes a set of OptionButton statements in a dialog definition block to define an option button group.

Syntax

```
OptionGroup .Identifier
```

...option button definition statements..

Parameters

.Identifier - name used as the identifier for the option button group. The name must start with the period character.

Notes

This statement is only used within a dialog box definition. It gives a name to a group of option or radio buttons which are placed immediately after it in the dialog definition.

The .Identifier parameter is used to reference the selected option in the group within the dialog object variable. For example, if myDlg is dimensioned as the variable to hold the value of the dialog controls, then a reference of the form:

myDlg.Identifier

will set or return an integer representing the number of the option button in the group which is checked. The first option button defined after the OptionGroup statement in the dialog definition is 0, the second is 1, etc.

Example

You can use option buttons to allow the user to choose one option from several options. Typically, you would use a group box to surround a group of option buttons, but you can also use a group box to set off a group of check boxes or any related group of controls.

```
Sub Main
 Begin Dialog GroupSample 31, 32, 185, 96, "Groups Example"
  OKButton 28, 68, 40, 14
  CancelButton 120, 68, 40, 14
  GroupBox 12, 8, 72, 52, "GroupBox", .GroupBox1
  GroupBox 100, 12, 72, 48, "GroupBox", .GroupBox2
  OptionGroup .OptionGroup1
  OptionButton 16, 24, 54, 8, "0ption 1"
  OptionButton 16, 40, 54, 8, "0ption 2"
  CheckBox 108, 24, 45, 8, "CheckBox 1", .CheckBox1
  CheckBox 108, 40, 45, 8, "CheckBox 2", .CheckBox2
 End Dialog
 Dim Dlg1 As GroupSample
 If Dialog ( Dlg1 ) Then
  Select Case Dlg1.OptionGroup1
   Case 0
    MsgBox "Option 1 was selected"
   Case 1
    MsgBox "Option 2 was selected"
  End Select
 End If
End Sub
```

PushButton

Description

Used in a dialog definition for placing a general-purpose pushbutton or control button in a user-defined dialog box.

Syntax

```
PushButton PosX, PosY, Width, Height, Caption$, .Identifier
```

Parameters

PosX,

Pos Y - numeric expressions representing the position of the top-left corner of the button expressed in pixels relative to the top-left corner of the dialog box.

Width - a numeric expression representing the width of the button in pixels.

Height - a numeric expression representing the height of the button in pixels.

Caption$ - a string expression containing the text to appear on the button.

.Identifier - name used as the identifier for the pushbutton control. The name must start with the period character.

Notes

This procedure is only used as part of a dialog definition.

Text

Description

This procedure is used within a dialog box definition to create a text field for titles and labels.

Syntax

```
Text PosX, PosY, Width, Height, Label$
```

Parameters

PosX, Pos Y - numeric expressions representing the position of the top-left corner of the control expressed in pixels relative to the top-left corner of the dialog box.

Width - a numeric expression representing the width of the control in pixels.

Height - a numeric expression representing the height of the control in pixels.

Label$ - a string expression that contains the text displayed.

Notes

Text placed in a dialog with this procedure is fixed and cannot be altered by the user. This statement is only used within a dialog definition.

Example

This example creates a dialog called DialogName1 with the title of "ASC - Hello".

```
Sub Main ()
 ' Define the dialog box.
 Begin Dialog DialogName1 60, 60, 160, 70, "ASC - Hello"
  TEXT 10, 10, 28, 12, "Name:"
  TEXTBOX 42, 10, 108, 12, .nameStr
  TEXTBOX 42, 24, 108, 12, .descStr
  CHECKBOX 42, 38, 48, 12, "&CHECKME", .checkInt
  OKBUTTON 42, 54, 40, 12
 End Dialog
 ' Dimension an object to represent the dialog.
 Dim Dlg1 As DialogName1
 Dlg1.checkInt = 1            ' Check the checkbox.
 Dialog Dlg1                  ' Display the dialog.
End Sub
```

TextBox

Description

This procedure is used within a dialog box definition to create a text box for data entry.

Syntax

```
TextBox PosX, PosY, Width, Height, .Identifier
```

Parameters

PosX, Pos Y - numbers or numeric expressions representing the position of the top-left corner of the control expressed in pixels relative to the top-left corner of the dialog box.

Width - a number or numeric expression representing the width of the control in pixels.

Height - a number or numeric expression representing the height of the control in pixels.

.Identifier - the identifier for the control. The identifier must begin with the period "." character.

Notes

A TextBox control has no inherent label associated with it. Use the Text statement to create a label if necessary.

The .Identifier parameter is used to return or set the text value of the TextBox. For example, if the object variable Dlg1 has be dimensioned to hold the values for the dialog box, then the statement Dlg1.Identifier = "my text" sets the text in TextBox. Similarly, A$ = Dlg1.Identifier returns the current text in the TextBox.

Example

This example creates a dialog called DialogName1 with the title of "ASC - Hello".

```
Sub Main ()
 ' Define the dialog box.
 Begin Dialog DialogName1 60, 60, 160, 70, "ASC - Hello"
  TEXT 10, 10, 28, 12, "Name:"
  TEXTBOX 42, 10, 108, 12, .nameStr
  TEXTBOX 42, 24, 108, 12, .descStr
  CHECKBOX 42, 38, 48, 12, "&CHECKME", .checkInt
  OKBUTTON 42, 54, 40, 12
 End Dialog
 ' Dimension an object to represent the dialog.
 Dim Dlg1 As DialogName1
 Dlg1.checkInt = 1            ' Check the checkbox.
 Dialog Dlg1                  ' Display the dialog.
End Sub
```

Overview of OLE support in Enable Basic

Object linking and embedding (OLE) is a technology that allows a programmer of Windows-based applications to create an application that can display data from many different applications and allows the user to edit that data from within the application in which it was created. In some cases, the user can even edit the data from within their application.

An OLE object refers to a discrete unit of data supplied by an OLE application. An application can expose many types of objects. For example, a spreadsheet application can expose a worksheet, macro sheet, chart, cell, or range of cells all as different types of objects. You use the OLE control to create linked and embedded objects. When a linked or embedded object is created, it contains the name of the application that supplied the object, its data (or, in the case of a linked object, a reference to the data) and an image of the data.

Some applications provide objects that support OLE Automation. You can use Enable Basic to programmatically manipulate the data in these objects. Some objects that support OLE Automation also support linking and embedding. You can create an OLE Automation object by using the `CreateObject` function.

An objects class determines the application that provides the object's data and the type of data the object contains. The class names of some commonly used Microsoft applications include MSGraph, MSDraw, WordDocument, and ExcelWorksheet.

In this section:

What is an OLE object?

An OLE Automation Object is an instance of a class within your application that you wish to manipulate programmatically, such as with Enable Basic. These may be new classes whose sole purpose is to collect and expose data and functions in a way that makes sense to your customers.

The object becomes programmable when you expose those member functions. OLE Automation defines two types of members that you may expose for an object:

- **Methods** are member functions that perform an action on an object. For example, a Document object might provide a Save method.
- **Properties** are member function pairs that set or return information about the state of an object. For example, a Drawing object might have a style property.

For example, Microsoft suggests the following objects could be exposed by implementing the listed methods and properties for each object:

OLE Automation object	Methods	Properties
Application	Help	ActiveDocument
	Quit	Application
	Add Data	Caption
	Repeat	DefaultFilePath
	Undo	Documents
		Height
		Name
		Parent
		Path
		Printers
		StatusBar
		Top

OLE Automation object	Methods	Properties
		Value
		Visible
		Width
Document	Activate	Application
	Close	Author
	NewWindow	Comments
	Print	FullName
	PrintPreview	Keywords
	RevertToSaved	Name
	Save	Parent
	SaveAs	Path
		ReadOnly
		Saved
		Subject
		Title
		Value

To provide access to more than one instance of an object, expose a collection object. A collection object manages other objects. All collection objects support iteration over the objects they manage. For example, Microsoft suggests an application with a multiple document interface (MDI) might expose a Documents collection object with the following methods and properties:

Collection object	Methods	Properties
Documents	Add	Application
	Close	Count
	Item	Parent
	Open	

OLE Automation and Enable Basic

Applications that support OLE Automation can be controlled by Enable Basic. You can use Enable Basic to manipulate objects by invoking methods on the object, or by getting and setting the object's properties, just as you would with the objects in Enable Basic. For example, if you created an OLE Automation object named MyObj, you might write code such as this to manipulate the object:

```
Sub Main
 Dim MyObj As Object
 Set MyObj = CreateObject ("Word.Basic")
 MyObj.FileNewDefault
 MyObj.Insert "Hello, world."
 MyObj.Bold 1
End Sub
```

The following functions and properties allow you to access an OLE Automation object:

- `CreateObject`: Creates a new object of a specified type.
- `GetObject`: Retrieves an object pointer to a running application

The following syntax is supported for the `GetObject` function:

```
Set MyObj = GetObject ("", class)
```

Where class is the parameter representing the class of the object to retrieve. The first parameter at this time must be an empty string.

The properties and methods an object supports are defined by the application that created the object. See that application's documentation for details on the properties and methods it supports.

Passing and Returning Strings to Enable Basic with OLE

The Macro server maintains variable-length strings internally as BSTRs.

BSTRs are defined in the OLE header files as OLECHAR FAR *. An OLECHAR is a UNICODE character in 32-bit OLE and an ANSI character in 16-bit OLE. A BSTR can contain NULL values because a length is also maintained with the BSTR. BSTRs are also NULL terminated so they can be treated as an LPSTR. Currently this length is stored immediately prior to the string. This may change in the future, so you should use the OLE APIs to access the string length.

You can pass a string from Chart to a DLL in one of two ways. You can pass it "by value" (`ByVal`) or "by reference" (`ByRef`). When you pass a string by value, Chart passes a pointer to the beginning of the string data (i.e. it passes a BSTR). When a string is passed by reference, Macro passes a pointer to a pointer to the string data (i.e. it passes a BSTR *).

OLE API

SysAllocString/SysAllocStringLen

SysAllocString/SysAllocStringLen

SysFreeString

SysStringLen

SysReAllocStringLen

SysReAllocString

Note

The BSTR is a pointer to the string, so you don't need to dereference it.

Overview of Server Process Support in Enable Basic

All servers in Altium Designer have processes stored in corresponding server install files with an `*.INS` extension. Server processes can be modelled at high levels of abstraction eg report electrical rules violations in the same style as low level system functions (for example obtaining a sheet handle of a document in the Altium Designer).

A parametric server process allows the information a process needs to be passed when the process is called. This ability to be able to pass process parameters allows direct control over the operation of a process.

Client, FPGA Flow, Integrated Library, PCB, Schematic and Workspace Manager processes are covered in the *Server Process Reference*.

Each server process has a process identifier. The process identifier is made up of two parts separated by a colon. The first part of the process identifier indicates the server that defines the process and the second part is the process name.

For example, the process `Sch:ZoomIn` is provided by Protel's Schematic server. When this process is launched, either by selecting a menu item, pressing a hot key or activating a toolbar button (which are all defined as process launchers in Altium Designer), it will perform the task of zooming in on the currently active schematic sheet.

Generally a process is executed by selecting a packaged process launcher (such as clicking on a toolbar button, or pressing a hot key or selecting a menu item) called as a command in Altium Designer, however you may wish to manually run a process. Up to three different types of process launchers can be used to launch the same process.

For parametric processes, each parameter has a value assigned and this parameter / value block is represented as Parameter = Name.

For example,

```
FileName = C:\Program Files\TestFile.Txt.
```

To concatenate several parameters as a whole string, each parameter / value block is separated by the pipe | symbol.

For example,

```
Parameter1 = Name1 | Parameter2 = Name 2 etc.
```

There are two ways you can execute a process in a script

To execute a server process in a script, you need to use Process Routines such as ResetParameters and RunProcess procedures or Client.SendMessage function.

Example 1

```
Sub SchJump
 ResetParameters     ' Clear parameter list
' The process "Sch:JumpNewLocation" requires 2 integer parameters -
' LocationX & LocationY - which set the location to jump to.
 AddIntegerParameter "Location.X", 100
 AddIntegerParameter "Location.Y", 100
 RunProcess "Sch:JumpNewLocation"
End Sub
```

Example 2

```
Client.SendMessage("WorkspaceManager:OpenObject","OpenMode=NewFromTemplate |
ObjectKind=Project",1024,Nil);
```

Server Processes Support

This section provides an overview of the following Server Processes Support:

AddColorParameter	*AddIntegerParameter*	*AddLongIntParameter*
AddSingleParameter	*AddStringParameter*	*GetColorParameter*
GetIntegerParameter	*GetLongIntParameter*	*GetSingleParameter*
GetStringParameter		

AddColorParameter

Description

Adds a color specification parameter to the parameter buffer.

Syntax

```
AddColorParameter ParamName$, RedVal, GreenVal, BlueVal
```

Parameters

ParamName$ - a string expression that represents the name of the parameter to set, usually "Color".

RedVal - integer expression from 0 to 255 representing the red level in an RGB color specification.

GreenVal - integer expression from 0 to 255 representing the green level in an RGB color specification.

BlueVal - integer expression from 0 to 255 representing the blue level in an RGB color specification.

Notes

This extension is used to define a color for use by a process that requires a color parameter. Running this extension constructs a parameter of the form:

Color = number

where number = RedVal + 256*(GreenVal + 256*BlueVal) and Color is the name contained in the ParamName$ parameter.

Example

When the following example is run with a schematic document active, it prompts the user to select a location and then draws a 200 unit long red wire from the cursor location.

```
Sub Main
 Dim AResult As Integer, X As Integer, Y As Integer
 ResetParameters           ' Reset the parameter buffer
 RunProcess "Sch:AskForXYLocation" ' Run a schematic process
```

```
' The AskForXYLocation process returns three parameters:
' LocationX, LocationY and Result. Result = 1 if the user
' selects a valid schematic location, otherwise it is 0.
 GetIntegerParameter "LocationX", X ' Get the X and Y values
 GetIntegerParameter "LocationY", Y ' of the location.
 GetIntegerParameter "Result", AResult ' Get the Result param.
 If AResult <> 0 then
' The PlaceWire process requires 5 pramameters: Location1X,
' Location1Y, Location2X, Location2Y and Color.
  ResetParameters           ' Reset the parameter buffer
  AddColorParameter "Color", 255,0,0   ' Set Color param.
  AddIntegerParameter "Location1.X", X ' Set X start position.
  AddIntegerParameter "Location1.Y", Y ' Set Y start position.
  AddIntegerParameter "Location2.X", X + 200 ' Set X end position.
  AddIntegerParameter "Location2.Y", Y ' Set Y end position.
  RunProcess "Sch:PlaceWire"           ' Place the wire.
 End If
 MsgBox "The wire has been drawn", 64, "Wire Me"
End Sub
```

AddIntegerParameter

Description

Adds a parameter with an Integer data type to the parameter buffer.

Syntax

```
AddIntegerParameter ParamName$, value
```

Parameters

ParamName$ - a string expression that represents the name of the process parameter.

value - an Integer expression that represents the value of the process parameter.

Notes

This extension is used to set an integer value for use by a process that requires an integer parameter whose value is between -32,768 and 32,767. Running this extension constructs a parameter of the form:

ValName = integer

where integer = value and ValName is the name contained in the ParamName$ parameter.

Example

When the following example is run with a schematic document active, it prompts the user to select a location and then draws a 200 unit long red wire from the cursor location.

```
Sub Main
 Dim AResult As Integer, X As Integer, Y As Integer
 ResetParameters           ' Reset the parameter buffer
 RunProcess "Sch:AskForXYLocation" ' Run a schematic process
' The AskForXYLocation process returns three parameters:
' LocationX, LocationY and Result. Result = 1 if the user
' selects a valid schematic location, otherwise it is 0.
 GetIntegerParameter "LocationX", X ' Get the X and Y values
 GetIntegerParameter "LocationY", Y ' of the location.
 GetIntegerParameter "Result", AResult ' Get the Result param.
 If AResult <> 0 then
```

```
' The PlaceWire process requires 5 pramameters: Location1X,
' Location1Y, Location2X, Location2Y and Color.
  ResetParameters           ' Reset the parameter buffer
  AddColorParameter "Color", 255,0,0   ' Set Color param.
  AddIntegerParameter "Location1.X", X ' Set X start position.
  AddIntegerParameter "Location1.Y", Y ' Set Y start position.
  AddIntegerParameter "Location2.X", X + 200 ' Set X end position.
  AddIntegerParameter "Location2.Y", Y ' Set Y end position.
  RunProcess "Sch:PlaceWire"           ' Place the wire.
 End If
 MsgBox "The wire has been drawn", 64, "Wire Me"
End Sub
```

AddLongIntParameter

Description

Adds a parameter with a Long integer data type to the parameter buffer.

Syntax

```
AddLongIntParameter ParamName$, value
```

Parameters

ParamName$ - a string expression that represents the name of the process parameter.

value - a Long integer expression that represents the value of the process parameter.

Notes

This extension is used to set an integer value for use by a process that requires an integer parameter whose value is greater than that stored by a standard Integer variable. Running this extension constructs a parameter of the form:

ValName = integer

where integer = value and ValName is the name contained in the ParamName$ parameter.

You can use this extension instead of AddIntegerParameter if you are unsure how large the value will be.

Example

When the following example is run with a schematic document active, it prompts the user to select a location and then draws a 200 unit long red wire from the cursor location.

```
Sub Main
 Dim AResult As Integer, X As Long, Y As Long
 ResetParameters           ' Reset the parameter buffer
 RunProcess "Sch:AskForXYLocation" ' Run a schematic process
' The AskForXYLocation process returns three parameters:
' LocationX, LocationY and Result. Result = 1 if the user
' selected a valid schematic location, otherwise it is 0.
 GetLongIntParameter "LocationX", X ' Get the X and Y values
 GetLongIntParameter "LocationY", Y ' of the location.
 GetIntegerParameter "Result", AResult ' Get the Result param.
 If AResult <> 0 then
' The PlaceWire process requires 5 pramameters: Location1X,
' Location1Y, Location2X, Location2Y and Color.
  ResetParameters           ' Reset the parameter buffer
  AddColorParameter "Color", 255,0,0   ' Set Color param.
  AddLongIntParameter "Location1.X", X ' Set X start position.
  AddLongIntParameter "Location1.Y", Y ' Set Y start position.
  AddLongIntParameter "Location2.X", X + 200 ' Set X end position.
```

```
  AddLongIntParameter "Location2.Y", Y ' Set Y end position.
  RunProcess "Sch:PlaceWire"           ' Place the wire.
 End If
 MsgBox "The wire has been drawn", 64, "Wire Me"
End Sub
```

Note

The use of the LongInt form of the extension is unnecessary in this macro as the parameter values will not exceed the bounds for a normal integer. They are used here for illustration purposes.

AddSingleParameter

Description

Adds a parameter with a single-precision value to the parameter buffer.

Syntax

```
AddSingleParameter ParamName$, value
```

Parameters

ParamName$ - a string expression that represents the name of the process parameter.

value - a Single precision expression that represents the value of the process parameter.

Notes

This extension is used to set a single-precision real value for use by a process that requires an Single data type number parameter. Running this extension constructs a parameter of the form:

ValName = number

where number = value and ValName is the name contained in the ParamName$ parameter.

AddStringParameter

Description

Adds a parameter with a string value to the parameter buffer.

Syntax

```
AddStringParameter ParamName$, value$
```

Parameters

ParamName$ - a string expression that represents the name of the process parameter.

value - a string expression that represents the value of the process parameter.

Notes

This extension is used to set a string for use by a process that requires a String data type value. Running this extension constructs a parameter of the form:

ValName = string

where string is the string contained in value$ and ValName is the name contained in the ParamName$ parameter.

Example

The following example constructs the string parameter Dialog = Color and adds it to the parameter buffer. The example then calls the Client: RunCommonDialog process, which uses the parameter and launches the Choose Color dialog.

```
Sub Main
 ResetParameters                  ' Reset the parameter buffer
 AddStringParameter "Dialog", "Color"
 RunProcess "Client:RunCommonDialog"
End Sub
```

GetColorParameter

Description

Retrieves the values of a color parameter from the return buffer after the running a process that returns a color.

Syntax

```
GetColorParameter ParamName$, RedVal, GreenVal, BlueVal
```

Parameters

ParamName$ - a string expression that contains the name of the parameter to retrieve, usually "Color".

RedVar - name of a valid integer variable to accept the value of the Red component of an RGB color description.

GreenVar - name of a valid integer variable to accept the value of the Green component of an RGB color description.

BlueVar - name of a valid integer variable to accept the value of the Blue component of an RGB color description.

Notes

You must run a process that returns a Color parameter before running this extension. If there is no Color parameter called ParamName$ in the return buffer, the extension will generate an error.

You must declare the RedVar, GreenVar and BlueVar variables as Integer data type variables before using this extension.

GetIntegerParameter

Description

Retrieves the value of an integer parameter from the return buffer after the running a process that returns an integer.

Syntax

```
GetIntegerParameter ParamName$, IntVar
```

Parameters

ParamName$ - a string expression that contains the name of the parameter to retrieve.

InVar - the name of a valid integer variable in which to store the value of the parameter.

Notes

You must run a process that returns an integer parameter before running this extension. If there is no integer parameter called ParamName$ in the return buffer, the extension will generate an error.

You must declare the IntVar variable as an Integer data type before using this extension.

Example

When the following example is run with a schematic document active, it prompts the user to select a location and then draws a 200 unit long red wire from the cursor location.

```
Sub Main
 Dim AResult As Integer, X As Integer, Y As Integer
 ResetParameters           ' Reset the parameter buffer
 RunProcess "Sch:AskForXYLocation" ' Run a schematic process
' The AskForXYLocation process returns three parameters:
' LocationX, LocationY and Result. Result = 1 if the user
' selects a valid schematic location, otherwise it is 0.
 GetIntegerParameter "LocationX", X ' Get the X and Y values
 GetIntegerParameter "LocationY", Y ' of the location.
 GetIntegerParameter "Result", AResult ' Get the Result param.
 If AResult <> 0 then
' The PlaceWire process requires 5 pramameters: Location1X,
' Location1Y, Location2X, Location2Y and Color.
  ResetParameters           ' Reset the parameter buffer
  AddColorParameter "Color", 255,0,0   ' Set Color param.
  AddIntegerParameter "Location1.X", X ' Set X start position.
  AddIntegerParameter "Location1.Y", Y ' Set Y start position.
  AddIntegerParameter "Location2.X", X + 200 ' Set X end position.
  AddIntegerParameter "Location2.Y", Y ' Set Y end position.
  RunProcess "Sch:PlaceWire"           ' Place the wire.
 End If
 MsgBox "The wire has been drawn", 64, "Wire Me"
```

```
End Sub
```

GetLongIntParameter

Description

Retrieves the value of a long integer parameter from the return buffer after the running a process that returns a long integer.

Syntax

```
GetLongIntParameter ParamName$, LongVar
```

Parameters

ParamName$ - a string expression that returns the name of the parameter to retrieve.

LongVar - the name of a valid long integer variable in which to store the value of the parameter.

Notes

You must run a process that returns an Integer or Long integer parameter before running this extension. If there is no integer or long integer parameter called ParamName$ in the return buffer, the extension will generate an error.

You must declare the LongVar variable as a Long data type before using this extension.

GetSingleParameter

Description

Retrieves the value of a real number parameter from the return buffer after the running a process that returns a real number.

Syntax

```
GetSingleParameter ParamName$, SingleVar
```

Parameters

ParamName$ - a string expression that returns the name of the parameter to retrieve.

SingleVar - the name of a valid single precision variable in which to store the value of the parameter.

Notes

You must run a process that returns a single precision parameter before running this extension. If there is no real number parameter called ParamName$ in the return buffer, the extension will generate an error.

You must declare the SingleVar variable as a Single data type before using this extension.

GetStringParameter

Description

Retrieves the value of a string parameter from the return buffer after the running a process that returns a string.

Syntax

```
GetStringParameter ParamName$, StringVar
```

Parameters

ParamName$ - a string expression that returns the name of the parameter to retrieve.

LongVar - the name of a valid string variable in which to store the value of the parameter.

Notes

You must run a process that returns a string parameter before running this extension. If there is no string parameter called ParamName$ in the return buffer, the extension will generate an error.

You must declare the StringVar variable as a String data type before using this extension.

Revision History

Date	Version No.	Revision
01-Dec-2004	1.0	New product release
26-Apr-2005	1.1	Updated for Altium Designer
15-Dec-2005	1.2	Updated for Altium Designer 6
4-Dec-2007	1.3	Updated for Altium Designer 6.9
27-Feb-2008	1.4	Updated for A4 page size.
30-Aug-2011	-	Updated template.