BasicScript 2.25
Language Reference

Summit Software
Confidential

September 25, 1996

Contents

Introduction

This manual provides a complete reference for the BasicScript 2.25 scripting language. It contains the following:

- The Language Summary provides you with a list of all functions, statements, and methods in the BasicScript language. These items are grouped by the task you wish to accomplish, so you can easily find the BasicScript language item that will help you do your work.
- The A-Z Reference provides detailed explanations of each item in the BasicScript language. It also provides concise descriptions of important topics.
- Appendix A, "Language Elements by Platform," provides a quick, alphabetic list of the items in the BasicScript language that also shows the platforms supported by each item.

Typographic Conventions

This manual uses the following typographic conventions.

Convention	Description
Do...Loop	Words in this typeface indicate elements of the BasicScript language.
variable	Words in italics indicate placeholders for parameters that you replace using the syntax described in this manual.
text$	In syntax, the presence of a type-declaration character following a parameter signifies that the parameter must be a variable of that type or an expression that evaluates to that type. If a parameter does not appear with a type-declaration character, then its type is described in the text.
[*expressionlist*]	Square brackets indicate that the enclosed items are optional.

Convention	Description

	Note: In BasicScript, you cannot end a statement with a comma, even if the parameters are optional:
	`MsgBox "Hello",,"Message"  'OK`
	`MsgBox "Hello",,           'Not valid`

{**Input** | **Binary**}	Braces indicate that you must choose one of the enclosed items, which are separated by a vertical bar.
...	Ellipses indicate that the preceding expression can be repeated any number of times.
`' Comment`	An apostrophe (') indicates the start of a comment.

Language Summary

The following table summarizes the functions, statements, methods and other items that belong to the BasicScript language. Items are grouped by the tasks you might wish to perform.

BasicScript Functions, Statements, and Methods by Category and Task

Category	Task	Language Element(s)
Arrays	Return the number of dimensions of an array	ArrayDims (function)
	Sort an array	ArraySort (statement)
	Erase the elements in one or more arrays	Erase (statement)
	Return the lower bound of a given array dimension	LBound (function)
	Change the default lower bound for array declarations	Option Base (statement)
	Re-establish the dimensions of an array	Redim (statement)
	Return the upper bound of a dimension of an array	UBound (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
BasicScript information	Return the CPU architecture of the current system	Basic.Architecture$ (property)
	Return the capabilities of the platform	Basic.Capability (method)
	Return the code page for the current locale	Basic.CodePage (property)
	Return the end-of-line character for the platform	Basic.Eoln$ (property)
	Return the available memory	Basic.FreeMemory (property)
	Return the directory where BasicScript is located	Basic.HomeDir$ (property)
	Return the locale of the current system	Basic.Locale$ (property)
	Return the name of the current operating system	Basic.OperatingSystem$ (property)
	Return the name of the vendor of the current operating system	Basic.OperatingSystemVendor$ (property)
	Return the version of the current operating system	Basic.OperatingSystemVersion$ (property)
	Return the platform id	Basic.OS (property)
	Return the path separator character for the platform	Basic.PathSeparator$ (property)
	Return the name of the CPU of the current system	Basic.Processor$ (property)
	Returns the number of CPUs installed on the current system	Basic.ProcessorCount (property)
	Return the version of BasicScript	Basic.Version$ (property)
Clipboard	Return the content of the clipboard as a string	Clipboard$ (function)
	Set the content of the clipboard	Clipboard$ (statement)
	Clear the clipboard	Clipboard.Clear (method)
	Get the type of data stored in the clipboard	Clipboard.GetFormat (method)
	Get text from the clipboard	Clipboard.GetText (method)
	Set the content of the clipboard to text	Clipboard.SetText (method)
Comments	Comment to end-of-line	Rem (statement)
	Add a comment	' (keyword)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Controlling other applications	Activate an application	AppActivate (statement)
	Close an application	AppClose (statement)
	Return the filename corresponding to an application	AppFilename$ (function)
	Return the full name of an application	AppFind, AppFind$ (functions)
	Return the name of the active application	AppGetActive$ (function)
	Get the position and size of an application	AppGetPosition (statement)
	Get the window state of an application	AppGetState (function)
	Hide an application	AppHide (statement)
	Fill an array with a list of running applications	AppList (statement)
	Maximize an application	AppMaximize (statement)
	Minimize an application	AppMinimize (statement)
	Move an application	AppMove (statement)
	Restore an application	AppRestore (statement)
	Set the state of an application's window	AppSetState (statement)
	Show an application	AppShow (statement)
	Change the size of an application	AppSize (statement)
	Return the type of an application	AppType (function)
	Simulate keystrokes in another application	DoKeys (statement)
	Send keystrokes to another application	SendKeys (statement)
	Execute another application	Shell (function)
Controlling menus in other applications	Execute a menu command in another application	Menu (statement)
	Determine if a menu item is checked in another application	MenuItemChecked (function)
	Determine if a menu item is enabled in another application	MenuItemEnabled (function)
	Determine if a menu item exists in another application	MenuItemExists (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Controlling windows in other applications	Activate a window	WinActivate (statement)
	Close a window	WinClose (statement)
	Find a window given its name	WinFind (function)
	Fill an array with window objects, one for each top-level window	WinList (statement)
	Change the size of a window	WinMaximize (statement), WinMinimize (statement), WinRestore (statement), WinSize (statement)
	Move a window	WinMove (statement)
	Scroll the active window left/right by a specified number of lines	HLine (statement)
	Scroll the active window left/right by a specified number of pages	HPage (statement)
	Scroll the active window left/right to a specified absolute position	HScroll (statement)
	Scroll the active window up/down by a specified number of lines	VLine (statement)
	Scroll the active window up/down by a specified number of pages	VPage (statement)
	Scroll the active window up/down to a specified absolute position	VScroll (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Conversion	Return the value of a character	Asc, AscB, AscW(functions)
	Convert one numeric value to another	CBool (function), CCur (function), CDate, CVDate (functions), CDbl (function), CInt (function), CLng (function), CSng (function), CStr (function), CVar (function), Fix (function), Int (function)
	Convert a character value to a string	Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ (functions)
	Convert a value to an error	CVErr (function)
	Convert a number to a hexadecimal string	Hex, Hex$ (functions)
	Determine if an expression is convertible to a date	IsDate (function)
	Determine if a variant contains a user-defined error value	IsError (function)
	Determine if an expression is convertible to a number	IsNumeric (function)
	Convert a number to an octal string	Oct, Oct$ (functions)
	Convert a number to a string	Str, Str$ (functions)
	Convert a string to a number	Val (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Date/time	Return the current date	Date, Date$ (functions)
	Change the system date	Date, Date$ (statements)
	Add a number of date intervals to a date	DateAdd (function)
	Subtract a number of date intervals from a date	DateDiff (function)
	Return a portion of a date	DatePart (function)
	Assemble a date from date parts	DateSerial (function)
	Convert a string to a date	DateValue (function)
	Return a component of a date value	Day (function), Hour (function), Minute (function), Month (function), Second (function), Weekday (function), Year (function)
	Return the current date and time	Now (function)
	Return the current system time	Time, Time$ (functions)
	Set the system time	Time, Time$ (statements)
	Return the number of elapsed seconds since midnight	Timer (function)
	Assemble a date/time value from time components	TimeSerial (function)
	Convert a string to a date/time value	TimeValue (function)
Desktop	Arrange the icons on the desktop	Desktop.ArrangeIcons (method)
	Cascades all non-minimized applications	Desktop.Cascade (method)
	Set the desktop colors	Desktop.SetColors (method)
	Set the desktop wallpaper	Desktop.SetWallpaper (method)
	Capture an image, placing it in the clipboard	Desktop.Snapshot (method)
	Tiles all non-minimized applications	Desktop.Tile (method)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Dialog manipulation	Activate a control	ActivateControl (statement)
	Determine if a control in another application's dialog is enabled	ButtonEnabled (function), CheckBoxEnabled (function), ComboBoxEnabled (function), EditEnabled (function), ListBoxEnabled (function), OptionEnabled (function)
	Determine if a control in another application's dialog exists	ButtonExists (function), CheckBoxExists (function), ComboBoxExists (function), EditExists (function), ListBoxExists (function), OptionExists (function)
	Retrieve a value from a control in another application's dialog box	GetCheckBox (function), GetComboBoxItem$ (function), GetComboBoxItemCount (function), GetEditText$ (function), GetListBoxItem$ (function), GetListBoxItemCount (function), GetOption (function)
	Select a control in another application's dialog box	SelectButton (statement), SelectComboBoxItem (statement), SelectListBoxItem (statement)
	Set the state of a control in another application's dialog box	SetCheckBox (statement), SetEditText (statement), SetOption (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Dynamic Data Exchange (DDE)	Execute a command in another application	DDEExecute (statement)
	Initiate a DDE conversation with another application	DDEInitiate (function)
	Set a value in another application	DDEPoke (statement)
	Return a value from another application	DDERequest, DDERequest$ (functions)
	Establish a DDE conversation, then set a value in another application	DDESend (statement)
	Terminate one or more conversations	DDETerminate (statement), DDETerminateAll (statement)
	Set the timeout used for non-responding applications	DDETimeout (statement)
Event queue	Empty a queue	QueEmpty (statement)
	Play back all events stored in a queue	QueFlush (statement)
	Add key down event to the queue	QueKeyDn (statement)
	Add key down/up events to the queue	QueKeys (statement)
	Add key up event to the queue	QueKeyUp (statement)
	Add mouse click to the queue	QueMouseClick (statement)
	Add mouse double-click to the queue	QueMouseDblClk (statement)
	Add mouse down/up/down events to the queue	QueMouseDblDn (statement)
	Add mouse down event to the queue	QueMouseDn (statement)
	Add mouse move event to the queue	QueMouseMove (statement)
	Add many mouse move events to the queue	QueMouseMoveBatch (statement)
	Add mouse up event to the queue	QueMouseUp (statement)
	Make all mouse positions in a queue relative to a window	QueSetRelativeWindow (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Error handling	Clear the properties of the Err object	Err.Clear (method)
	Set or retrieve the description of the Err object	Err.Description (property)
	Set or retrieve the help context ID of the Err object	Err.HelpContext (property)
	Set or retrieve the help file assocated with the Err object	Err.HelpFile (property)
	Return the last error generated by a call to a DLL	Err.LastDLLError (property)
	Return or set the number of the Err object	Err.Number (property)
	Generate a runtime error	Err.Raise (method)
	Set or retrieve the source of a runtime error	Err.Source (property)
	Return the line with the error	Erl (function)
	Simulate a trappable runtime error	Error (statement)
	Return the text of a given error	Error, Error$ (functions)
	Trap an error	On Error (statement)
	Continue execution after an error trap	Resume (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
File I/O	Close one or more files	Close (statement)
	Determine if the end-of-file has been reached	EOF (function)
	Return the next available file number	FreeFile (function)
	Read data from a random or binary file	Get (statement)
	Read data from a sequential file into variables	Input# (statement)
	Read a specified number of bytes from a file	Input, Input$, InputB, InputB$ (functions)
	Read a line of text from a sequential file	Line Input# (statement)
	Return the record position of the file pointer within a file	Loc (function)
	Lock or unlock a section of a file	Lock, Unlock (statements)
	Return the number of bytes in an open file	Lof (function)
	Open a file for reading or writing	Open (statement)
	Print data to a file	Print# (statement)
	Write data to a binary or random file	Put (statement)
	Close all open files	Reset
	Return the byte position of the file pointer within a file	Seek (function)
	Set the byte position of the file pointer which a file	Seek (statement)
	Specify the line width for sequential files	Width# (statement)
	Write data to a sequential file	Write# (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
File system	Change the current directory	ChDir (statement)
	Change the current drive	ChDrive (statement)
	Return the current directory	CurDir, CurDir$ (functions)
	Return files in a directory	Dir, Dir$ (functions)
	Fill an array with valid disk drive letters	DiskDrives (statement)
	Return the free space on a given disk drive	DiskFree (function)
	Return the mode in which a file is open	FileAttr (function)
	Copy a file	FileCopy (statement)
	Return the date and time when a file was last modified	FileDateTime (function)
	Fill an array with a subdirectory list	FileDirs (statement)
	Determine if a file exists	FileExists (function)
	Return the length of a file in bytes	FileLen (function)
	Fill an array with a list of files	FileList (statement)
	Return a portion of a filename	FileParse$ (function)$
	Return the type of a file	FileType (function)
	Return the attributes of a file	GetAttr (function)
	Delete files from disk	Kill (statement)
	Return a value representing a collection of same-type files on the Macintosh	MacID (function)
	Create a subdirectory	MkDir (statement)
	Rename a file	Name (statement)
	Remove a subdirectory	RmDir (statement)
	Change the attributes of a file	SetAttr (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Financial	Return depreciation of an asset using double-declining balance method	DDB (function)
	Return the future value of an annuity	Fv (function)
	Return the interest payment for a given period of an annuity	IPmt (function)
	Return the internal rate of return for a series of payments and receipts	IRR (function)
	Return the modified internal rate of return	MIRR (function)
	Return the number of periods of an annuity	NPer (function)
	Return the net present value of an annuity	Npv (function)
	Return the payment for an annuity	Pmt (function)
	Return the principal payment for a given period of an annuity	PPmt (function)
	Return the present value of an annuity	Pv (function)
	Return the interest rate for each period of an annuity	Rate (function)
	Return the straight-line depreciation of an asset	Sln (function)
	Return the Sum of Years' Digits depreciation of an asset	SYD (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Flow Control	Call a subroutine	Call (statement)
	Return a value at a given index	Choose (function)
	Execute a group of statements repeatedly	Do...Loop (statement)
	Yield control to other applications	DoEvents (function),DoEvents (statement)
	Stop execution of a script	End (statement)
	Exit a Do loop	Exit Do (statement)
	Exit a For loop	Exit For (statement)
	Execute a block of statements repeatedly	For...Next (statement)
	For Each...Next	For Each...Next (statement)
	Execute at a specific label, allowing control to return later	GoSub (statement)
	Execute at a specific label	Goto (statement)
	Conditionally execute one or more statements	If...Then...Else (statement)
	Return one of two values depending on a condition	IIf (function)
	Define a subroutine where execution begins	Main (statement)
	Continue execution after the most recent GoSub	Return (statement)
	Execute one of a series of statements	Select...Case (statement)
	Pause for a specified number of milliseconds	Sleep (statement)
	Suspend execution, returning to a debugger (if present)	Stop (statement)
	Return one of a series of expressions depending on a condition	Switch (function)
	Repeat a group of statements while a condition is True	While...Wend (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
INI Files and Registry	Delete a setting from the system registry or an INI file	DeleteSetting (statement)
	Return the values of all keys or settings within the system registry	GetAllSettings (function)
	Return the value of a key or setting within the system registry	GetSetting (function)
	Read a string from an INI file	ReadIni$ (function)
	Read all the item names from a given section of an INI file	ReadIniSection (statement)
	Update the value of a key or setting within the system registry	SaveSetting (statement)
	Write a new value to an INI file	WriteIni (statement)
Logical/binary operators	Perform logical or binary operations on two expressions	And (operator), Eqv (operator), Imp (operator), Not (operator), Or (operator), Xor (operator)
Math	Return the absolute value of a number	Abs (function)
	Return the arc tangent of a number	Atn (function)
	Return the cosine of an angle	Cos (function)
	Return e raised to a given power	Exp (function)
	Return the integer part of a number	Fix (function)
	Return the integer portion of a number	Int (function)
	Return the natural logarithm of a number	Log (function)
	Return a random number between two values	Random (function)
	Initialize the random number generator	Randomize (statement)
	Generate a random number between 0 and 1	Rnd (function)
	Return the sign of a number	Sgn (function)
	Return the sine of an angle	Sin (function)
	Return the square root of a number	Sqr (function)
	Return the tangent of an angle	Tan (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Miscellaneous	Define a preprocessor constant for the BasicScript compiler	#Const (directive)
	Direct the BasicScript compiler to include or exclude sections of code based on conditions	#If...Then...#Else (directive)
	Force parts of an expression to be evaluated before others	() (keyword)
	Add a line continuation character	_ (keyword)
	Make a sound	Beep (statement)
	Return the status of the Input Method Editor	IMEStatus (function)
	Allow execution or interpretation of a block of text	Inline (statement)
	Execute an AppleScript script	MacScript (statement)
	Execute an MCI command	Mci (function)
	Set the default data type of variables and return values	Option Default (statement)
	Prevent implicit delcarations of variables and return values	Option Explicit (statement)
	Print a file using the application to which the file belongs	PrintFile (function)
Network	Redirect a local device to a shared device on a network	Net.AddCon (method)
	Display a dialog requesting a network directory or printer resource	Net.Browse$ (method)
	Cancel a network connection	Net.CancelCon (method)
	Display a dialog allowing configuration of the network	Net.Dialog (method)
	Return information about the capabilities of the network	Net.GetCaps (method)
	Return the name of the network resource associated with a local device	Net.GetCon$ (method)
	Return the name of the user on the network	Net.User$ (method)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Numeric operators	Multiply	* (operator)
	Add	+ (operator)
	Subtract	- (operator)
	Divide	/ (operator)
	Integer divide	\ (operator)
	Raise to a power	^ (operator)
	Determine the remainder	Mod (operator)
Objects	Instantiate an OLE automation object	CreateObject (function)
	Return an OLE automation object from a file, or returns a previously instantiated OLE automation object	GetObject (function)
	Compare two object variables	Is (operator)
	Value indicating no valid object	Nothing
Open Database Connectivity (ODBC)	Specify where to place results with SQLRetrieve	SQLBind (function)
	Close a connection to a database	SQLClose (function)
	Return error information when an SQL function fails	SQLError (function)
	Execute a query against a database and return the number of rows or columns affected by the query	SQLExecQuery (function)
	Return information about the structure of a database	SQLGetSchema (function)
	Establishes a connection with a database	SQLOpen (function)
	Run a query against a database, returning the results as an array	SQLRequest (function)
	Retrieve all or part of a query	SQLRetrieve (function)
	Place the results of a query in a file	SQLRetrieveToFile (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Operating environment	Return the command line	Command, Command$ (functions)
	Return the operating system value of a window	HWND.Value (property)
	Return the value of an environment variable	Environ, Environ$ (functions)
	Return the free memory in the operating environment	System.FreeMemory (property)
	Return the free resources in the operating environment	System.FreeResources (property)
	Return the total available memory in the operating environment	System.TotalMemory (property)
	Return the directory containing Windows	System.WindowsDirectory$ (property)
	Return the Windows version	System.WindowsVersion$ (property)
	Exit the operating environment	System.Exit (method)
	Toggle mouse trails on or off	System.MouseTrails (method)
	Restart the operating environment	System.Restart (method)
Parsing	Return a range of items from a string	Item$ (function)
	Return the number of items in a string	ItemCount (function)
	Retrieve a line from a string	Line$ (function)
	Return the number of lines in a string	LineCount (function)
	Return a sequence of words from a string	Word$ (function)
	Return the number of words in a string	WordCount (function)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Predefined dialogs	Display a dialog asking a question	AnswerBox (function)
	Display a dialog allowing the user to type a response	AskBox, AskBox$ (functions)
	Display a dialog allowing the user to type a password	AskPassword, AskPassword$ (functions)
	Display a dialog allowing the user to type a response	InputBox, InputBox$ (functions)
	Display a dialog containing a message and some buttons	MsgBox (function)
	Display a dialog containing a message and some buttons	MsgBox (statement)
	Close a modeless message box	Msg.Close (method)
	Open a modeless message box	Msg.Open (method)
	Set the message contained within a modeless message box	Msg.Text (property)
	Set the percentage of the thermometer in a modeless message box	Msg.Thermometer (property)
	Display a dialog requesting a file to open	OpenFileName$ (function)
	Display a popup menu containing items from an array	PopupMenu (function)
	Display a dialog requesting the name of a new file	SaveFileName$ (function)
	Display a dialog allowing selection of an item from an array	SelectBox (function)
Printer	Retrieve the current printer orientation	PrinterGetOrientation (function)
	Set the printer orientation	PrinterSetOrientation (statement)
Printing	Print data to the screen	Print (statement)
	Print a number of spaces within a Print statement	Spc (function)
	Used with Print to print spaces up to a column position	Tab (function)
Procedures	Define an external routine or a forward reference	Declare (statement)
	Exit a function	Exit Function (statement)
	Exit a subroutine	Exit Sub (statement)
	Create a user-defined function	Function...End Function (statement)
	Create a user-defined subroutine	Sub...End Sub (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Screen resolution	Return the x dialog base units	Screen.DlgBaseUnitsX (property)
	Return the y dialog base units	Screen.DlgBaseUnitsY (property)
	Return the height of the display, in pixels	Screen.Height (property)
	Return the number of twips per pixel in the x direction	Screen.TwipsPerPixelX (property)
		Screen.TwipsPerPixelY (property)
	Return the number of twips per pixel in the y direction	Screen.Width (property)
	Return the width of the display, in pixels	

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Strings	Concetenate two strings	& (operator)
	Return a string formatted to a given specification	Format, Format$ (functions)
	Return the position of one string within another	InStr, InstrB (functions)
	Convert a string to lower case	LCase, LCase$ (functions)
	Return the left portion of a string	Left, Left$, LeftB, LeftB$ (functions)
	Return the length of a string or the size of a data item	Len, LenB (functions)
	Compare a string against a pattern	Like (operator)
	Left align a string or user-defined type within another	LSet (statement)
	Remove leading spaces from a string	LTrim, LTrim$ (functions)
	Return a substring from a string	Mid, Mid$, MidB, MidB$ (functions)
	Replace one part of a string with another	Mid, Mid$, MidB, MidB$ (statements)
	Change the default comparison between text and binary	Option Compare (statement)
	Allow interpretation of C-style escape sequences in strings	Option CStrings (statement)
	Return the right portion of a string	Right, Right$, RightB, RightB$ (functions)
	Right align a string within another	RSet (statement)
	Remove trailing spaces from a string	RTrim, RTrim$ (functions)
	Return a string os spaces	Space, Space$ (functions)
	Compare two strings	StrComp (function)
	Convert a string based on a conversion parameter	StrConv (function)
	Return a string consisting of a repeated character	String, String$ (functions)
	Trim leading and trailing spaces from a string	Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$ (functions)
	Return the upper case of a string	UCase, UCase$ (functions)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
User dialogs	Begin definition of a dialog template	Begin Dialog (statement)
	Add a control to a dialog box template	CancelButton (statement), CheckBox (statement), ComboBox (statement), DropListBox (statement), GroupBox (statement), ListBox (statement), OKButton (statement), OptionButton (statement), OptionGroup (statement), Picture (statement), PictureButton (statement), PushButton (statement), Text (statement), TextBox (statement)
	Invoke a user-dialog, returning which button was selected	Dialog (function)
	Invoke a user-dialog	Dialog (statement)
	Return the caption of the dynamic dialog	DlgCaption (function)
	Change the caption of the current dialog	DlgCaption (statement)
	Return the id of a control in a dynamic dialog	DlgControlId (function)
	Determine if a control is enabled in a dynamic dialog	DlgEnable (function)
	Enable or disables a control in a dynamic dialog	DlgEnable (statement)
	Return the control with the focus in a dynamic dialog	DlgFocus (function)
	Set focus to a control in a dynamic dialog	DlgFocus (statement)
	Set the content of a list box or combo box in a dynamic dialog	DlgListBoxArray (statement)
	Set the picture of a control in a dynamic dialog	DlgSetPicture (statement)
	Set the content of a control in a dynamic dialog	DlgText (statement)
	Return the content of a control in a dynamic dialog	DlgText$ (function)
	Return the value of a control in a dynamic dialog	DlgValue (function)
	Set the value of a control in a dynamic dialog	DlgValue (statement)
	Determine if a control is visible in a dynamic dialog	DlgVisible (function)
	Set the visibility of a control in a dynamic dialog	DlgVisible (statement)

BasicScript Functions, Statements, and Methods by Category and Task (Continued)

Category	Task	Language Element(s)
Variables and constants	Assignment	= (statement)
	Define a constant	Const (statement)
	Set the default data type	DefType (statement)
	Declare a local variable	Dim (statement)
	Declare variables for sharing between scripts	Global (statement)
	Assign a value to a variable	Let (statement)
	Declare variables accessible to all routines in a script	Private (statement)
	Declare variables accessible to all routines in all scripts	Public (statement)
	Assign an object variable	Set (statement)
	Declare a user-defined data type	Type (statement)
Variants	Determine if a variant has been initialized	IsEmpty (function)
	Determine if a variant contains a user-defined error	IsError (function)
	Determine if an optional parameter was specified	IsMissing (function)
	Determine if a variant contains valid data	IsNull (function)
	Determine if an expression contains an object	IsObject (function)
	Return the type of data stored in a variant	VarType (function)
Viewport	Clear the contents of the viewport	Viewport.Clear (method)
	Close the viewport	Viewport.Close (method)
	Open a viewport	Viewport.Open (method)

A-Z Reference

' (keyword)

Syntax `'`*text*

Description Causes the compiler to skip all characters between this character and the end of the current line.

Comments This is very useful for commenting your code to make it more readable.

Example

```
Sub Main()
      'This whole line is treated as a comment.
      i$="Strings"   'This is a valid assignment with a comment.
      This line will cause an error (the apostrophe is missing).
End Sub
```

See Also Rem (statement); Comments (topic).

Platform(s) All.

- (operator)

Syntax 1 *expression1* `-` *expression2*

Syntax 2 `-`*expression*

Description Returns the difference between *expression1* and *expression2* or, in the second syntax, returns the negation of *expression*.

Comments

Syntax 1

The type of the result is the same as that of the most precise expression, with the following exceptions:

If one expression is	and the other expression is	then the type of the result is
Long	**Single**	**Double**
Boolean	**Boolean**	**Integer**
Date	Date	Date
Date	any other data type	Double

A runtime error is generated if the result overflows its legal range.

When either or both expressions are **Variant**, then the following additional rules apply:

- If either expression is **Null**, then the result is **Null**.
- **Empty** is treated as an **Integer** of value 0.
- If the type of the result is an **Integer** variant that overflows, then the result is a **Long** variant.
- If the type of the result is a **Long**, **Single**, or **Date** variant that overflows, then the result is a **Double** variant.

Syntax 2

If *expression* is numeric, then the type of the result is the same type as *expression*, with the following exception:

- If *expression* is **Boolean**, then the result is **Integer**.

Note: In 2's complement arithmetic, unary minus may result in an overflow with **Integer** and **Long** variables when the value of *expression* is the largest negative number representable for that data type. For example, the following generates an overflow error:

```
Sub Main()
   Dim a As Integer
   a = -32768
   a = -a            'Generates overflow here.
End Sub
```

When negating variants, overflow will never occur because the result will be automatically promoted: integers to longs and longs to doubles.

Example

```
'This example assigns values to two numeric variables and
'their difference to a third variable, then displays the
'result.
Sub Main()
      i% = 100
      j# = 22.55
      k# = i% - j#
      MsgBox "The difference is: " & k#
End Sub
```

See Also Operator Precedence (topic).

Platform(s) All.

#Const (directive)

Syntax `#Const` *constname* `=` *expression*

Description Defines a preprocessor constant for use in the **#If...Then...#Else** statement.

Comments Internally, all preprocessor constants are of type **Variant**. Thus, the *expression* parameter can be any type.

Variables defined using **#Const** can only be used within the **#If...Then...#Else** statement and other **#Const** statements. Use the **Const** statement to define constants that can be used within your code.

Example

```
#Const SUBPLATFORM = "NT"
#Const MANUFACTURER = "Windows"
#Const TYPE = "Workstation"
#Const PLATFORM = MANUFACTURER & " " & SUBPLATFORM & " " & TYPE
Sub Main()
   #If PLATFORM = "Windows NT Workstation" Then
      MsgBox "Running under Windows NT Workstation"
   #End If
End Sub
```

See Also #If...Then...#Else (directive); Const (statement).

Platform(s) All.

#If...Then...#Else (directive)

Syntax

```
#If expression Then
   [statements]
[#ElseIf expression Then
   [statements]]
[#Else
   [statements]]
#End If
```

Description Causes the compiler to include or exclude sections of code based on conditions.

Comments The *expression* represents any valid BasicScript Boolean expression evaluating to **True** of **False**. The *expression* may consist of literals, operators, constants defined with **#Const**, and any of the following predefined constants:

Constant	Value
AIX	True if development environment is AIX.
HPUX	True if development environment is HPUX.

Constant	Value
Irix	True if development environment is Irix.
LINUX	True if development environment is LINUX.
Macintosh	True if development environment is Macintosh (68K or PowerPC).
MacPPC	True if development environment is PowerMac.
Mac68K	True if development environment is 68K Macintosh.
Netware	True if development environment is NetWare.
OS2	True if development environment is OS/2.
OSF1	True if development environment is OSF/1.
SCO	True if development environment is SCO.
Solaris	True if development environment is Solaris.
SunOS	True if development environment is SunOS.
UNIX	True if development environment is any UNIX platform.
UnixWare	True if development environment is UnixWare.
VMS	True if development environment is VMS.
Win16	True if development environment is 16-bit Windows.
Win32	True if development environment is 32-bit Windows.
Empty	**Empty**
False	**False**
Null	**Null**
True	**True**

The expression can use any of the following operators: +, -, *****, /, \, ^, + (unary), - (unary), **Mod**, **&**, =, <>, >=, >, <=, <, **And**, **Or**, **Xor**, **Imp**, **Eqv**.

If the *expression* evaluates to a numeric value, then it is considered **True** if non-zero, **False** if zero. If the expression evaluates to **String** not convertible to a number or evaluates to **Null**, then a "Type mismatch" error is generated.

Text comparisons within *expression* are always case-insensitive, regardless of the **Option Compare** setting

You can define your own constants using the **#Const** directive, and test for these constants within the *expression* parameter as shown below:

```
#Const VERSION = 2
Sub Main
   #If VERSION = 1 Then
      directory$ = "\apps\widget"
```

```
   #ElseIf VERSION = 2 Then
      directory$ = "\apps\widget32"
   #Else
      MsgBox "Unknown version."
   #End If
End Sub
```

Any constant not already defined evaluates to **Empty**.

A common use of the **#If...Then...#Else** directive is to optionally include debugging statements in your code. The following example shows how debugging code can be conditionally included to check parameters to a function:

```
#Const DEBUG = 1
Sub ChangeFormat(NewFormat As Integer,StatusText As String)
   #If DEBUG = 1 Then
      If NewFormat <> 1 And NewFormat <> 2 Then
         MsgBox "Parameter ""NewFormat"" is invalid."
         Exit Sub
      End If
      If Len(StatusText) > 78 Then
         MsgBox "Parameter ""StatusText"" is too long."
         Exit Sub
      End If
   #End If
   Rem Change the format here...
End Sub
```

Excluded section are not compiled by BasicScript, allowing you to exclude sections of code that has errors or doesn't even represent valid BasicScript syntax. For example, the following code uses the **#If...Then...#Else** statement to include a multi-line comment:

```
Sub Main
   #If 0
      The following section of code displays
      a dialog box containing a message and an
      OK button.
   #End If
   MsgBox "Hello, world."
End Sub
```

In the above example, since the expression **#If 0** never evaluates to True, the text between that and the matching **#End If** will never be compiled.

Example

```
'The following example calls an external routine. Calling
'External routines is very specific to the platform--thus,
'we have different code for each platform.
```

```
#If Win16 Then
   Declare Sub GetWindowsDirectory Lib "KERNEL" (ByVal _
      DirName As String,ByVal MaxLen As Integer)
#ElseIf Win32 Then
   Declare Sub GetWindowsDirectory Lib "KERNEL32" Alias _
      "GetWindowsDirectoryA" (ByVal DirName As String,ByVal _
      MaxLen As Long)
#End If
Sub Main()
   Dim DirName As String * 256
   GetWindowsDirectory DirName,len(DirName)
   MsgBox "Windows directory = " & DirName
End Sub
```

See Also #Const (directive).

Platform(s) All.

& (operator)

Syntax *expression1* & *expression2*

Description Returns the concatenation of *expression1* and *expression2*.

Comments If both expressions are strings, then the type of the result is **String**. Otherwise, the type of the result is a **String** variant.

When nonstring expressions are encountered, each expression is converted to a **String** variant. If both expressions are **Null**, then a **Null** variant is returned. If only one expression is **Null**, then it is treated as a zero-length string. **Empty** variants are also treated as zero-length strings.

In many instances, the plus (+) operator can be used in place of **&**. The difference is that + attempts addition when used with at least one numeric expression, whereas **&** always concatenates.

Example

```
'This example assigns a concatenated string to variable s$ and
'a string to s2$, then concatenates the two variables and
'displays the result in a dialog box.
Sub Main()
   s$ = "This string" & " is concatenated"
   s2$ = " with the & operator."
   MsgBox s$ & s2$
End Sub
```

See Also + (operator); Operator Precedence (topic).

Platform(s) All.

() (keyword)

Syntax 1 `...(`*expression*`)...`

Syntax 2 `...,(`*parameter*`),...`*Description*

Comments

Parentheses within Expressions

Parentheses override the normal precedence order of BasicScript operators, forcing a subexpression to be evaluated before other parts of the expression. For example, the use of parentheses in the following expressions causes different results:

```
i = 1 + 2 * 3        'Assigns 7.
i = (1 + 2) * 3      'Assigns 9.
```

Use of parentheses can make your code easier to read, removing any ambiguity in complicated expressions.

Parentheses Used in Parameter Passing

Parentheses can also be used when passing parameters to functions or subroutines to force a given parameter to be passed by value, as shown below:

```
ShowForm i        'Pass i by reference.
ShowForm (i)      'Pass i by value.
```

Enclosing parameters within parentheses can be misleading. For example, the following statement appears to be calling a function called **ShowForm** without assigning the result:

```
ShowForm(i)
```

The above statement actually calls a subroutine called **ShowForm**, passing it the variable **i** by value. It may be clearer to use the **ByVal** keyword in this case, which accomplishes the same thing:

```
ShowForm ByVal i
```

Note: The result of an expression is always passed by value.

Example

```
'This example uses parentheses to clarify an expression.
Sub Main()
   bill = False
   dave = True
   jim = True
   If (dave And bill) Or (jim And bill) Then
      MsgBox "The required parties for the meeting are here."
   Else
      MsgBox "Someone is late again!"
   End If
End Sub
```

See Also ByVal (keyword); Operator Precedence (topic).

Platform(s) All.

* (operator)

Syntax *expression1* * *expression2*

Description Returns the product of *expression1* and *expression2*.

Comments The result is the same type as the most precise expression, with the following exceptions:

If one expression is	and the other expression is	then the type of the result is
Single	**Long**	**Double**
Boolean	**Boolean**	**Integer**
Date	**Date**	**Double**

When the ***** operator is used with variants, the following additional rules apply:

- **Empty** is treated as 0.
- If the type of the result is an **Integer** variant that overflows, then the result is automatically promoted to a **Long** variant.
- If the type of the result is a **Single**, **Long**, or **Date** variant that overflows, then the result is automatically promoted to a **Double** variant.
- If either expression is **Null**, then the result is **Null**.

Example

```
'This example assigns values to two variables and their product
'to a third variable, then displays the product of s# * t#.
Sub Main()
   s# = 123.55
   t# = 2.55
   u# = s# * t#
   MsgBox s# & " * " & t# & " = " & s# * t#
End Sub
```

See Also Operator Precedence (topic).

Platform(s) All.

. (keyword)

Syntax 1 *object*.*property*

Syntax 2 *structure*.*member*

Description Separates an object from a property or a structure from a structure member.

Examples

```
'This example uses the period to separate an object from a
'property.
Sub Main()
      MsgBox Clipboard.GetText()
End Sub
'This example uses the period to separate a structure from a
'member.
Type Rect
      left As Integer
      top As Integer
      right As Integer
      bottom As Integer
End Type
Sub Main()
      Dim r As Rect
      r.left = 10
      r.right = 12
End Sub
```

See Also Objects (topic).

Platform(s) All.

/ (operator)

Syntax *expression1 / expression2*

Description Returns the quotient of *expression1* and *expression2*.

Comments The type of the result is **Double**, with the following exceptions:

If one expression is	and the other expression is	then the type of the result is
Integer	**Integer**	**Single**
Single	**Single**	**Single**
Boolean	**Boolean**	**Single**

A runtime error is generated if the result overflows its legal range.

When either or both expressions is **Variant**, then the following additional rules apply:

- If either expression is **Null**, then the result is **Null**.
- **Empty** is treated as an **Integer** of value 0.
- If both expressions are either **Integer** or **Single** variants and the result overflows, then the result is automatically promoted to a **Double** variant.

Example

```
'This example assigns values to two variables and their
'quotient to a  third variable, then displays the result.
Sub Main()
      i% = 100
      j# = 22.55
      k# = i% / j#
      MsgBox "The quotient of i/j is: " & k#
End Sub
```

See Also \ (operator); Operator Precedence (topic).

Platform(s) All.

\ (operator)

Syntax *expression1* `\` *expression2*

Description Returns the integer division of *expression1* and *expression2*.

Comments Before the integer division is performed, each expression is converted to the data type of the most precise expression. If the type of the expressions is either **Single**, **Double**, **Date**, or **Currency**, then each is rounded to **Long**.

If either expression is a **Variant**, then the following additional rules apply:

- If either expression is **Null**, then the result is **Null**.
- **Empty** is treated as an **Integer** of value 0.

Example

```
'This example assigns the quotient of two literals to a variable
'and displays the result.
Sub Main()
      s% = 100.99 \ 2.6
      MsgBox "Integer division of 100.99\2.6 is: " & s%
End Sub
```

See Also \ (operator); Operator Precedence (topic).

Platform(s) All.

^ (operator)

Syntax *expression1* `^` *expression2*

Description Returns *expression1* raised to the power specified in *expression2*.

Comments The following are special cases:

Special Case	Value
n^0	1
0^-n	Undefined
0^+n	0
1^n	1

The type of the result is always **Double**, except with **Boolean** expressions, in which case the result is **Boolean**. Fractional and negative exponents are allowed.

If either expression is a **Variant** containing **Null**, then the result is **Null**.

It is important to note that raising a number to a negative exponent produces a fractional result.

Example

```
Sub Main()
      s# = 2 ^ 5     'Returns 2 to the 5th power.
      r# = 16 ^ .5   'Returns the square root of 16.
      MsgBox "2 to the 5th power is: " & s#
      MsgBox "The square root of 16 is: " & r#
End Sub
```

See Also Operator Precedence (topic).

Platform(s) All.

_ (keyword)

Syntax *text1* _

text2

Description Line-continuation character, which allows you to split a single BasicScript statement onto more than one line.

Comments The line-continuation character cannot be used within strings and must be preceded by white space (either a space or a tab).

The line-continuation character can be followed by a comment, as shown below:

```
   i = 5 + 6 & _        'Continue on the next line.
         "Hello"
```

Example

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   'The line-continuation operator is useful when concatenating
   'long strings.
   message = "This is a line of text that" + crlf + "extends"  _
```

```
        + "beyond the borders of the editor" + crlf + "so it" _
        + "is split into multiple lines"
    'It is also useful for separating and continuing long
    'calculation lines.
    b# = .124
    a# = .223
    s# = ( (((Sin(b#) ^ 2) + (Cos(a#) ^ 2)) ^ .5) / _
          (((Sin(a#) ^ 2) + (Cos(b#) ^ 2)) ^ .5) ) * 2.00
    MsgBox message & crlf & "The value of s# is: " & s#
End Sub
```

Platform(s) All.

+ (operator)

Syntax *expression1* + *expression2*

Description Adds or concatenates two expressions.

Comments Addition operates differently depending on the type of the two expressions:

If one expression is	and the other expression is	then
Numeric	**Numeric**	Perform a numeric add (see below).
String	**String**	Concatenate, returning a string.
Numeric	**String**	A runtime error is generated.
Variant	**String**	Concatenate, returning a **String** variant.
Variant	**Numeric**	Perform a variant add (see below).
Empty variant	**Empty** variant	Return an **Integer** variant, value 0.
Empty variant	Any data type	Return the non-**Empty** operand unchanged.
Null variant	Any data type	Return **Null**.
Variant	**Variant**	Add if either is numeric; otherwise, concatenate.

When using **+** to concatenate two variants, the result depends on the types of each variant at runtime. You can remove any ambiguity by using the **&** operator.

Numeric Add

A numeric add is performed when both expressions are numeric (i.e., not variant or string). The result is the same type as the most precise expression, with the following exceptions:

If one expression is	and the other expression is	then the type of the result is
Single	**Long**	**Double**
Boolean	**Boolean**	**Integer**

A runtime error is generated if the result overflows its legal range.

Variant Add

If both expressions are variants, or one expression is **Numeric** and the other expression is **Variant**, then a variant add is performed. The rules for variant add are the same as those for normal numeric add, with the following exceptions:

- If the type of the result is an **Integer** variant that overflows, then the result is a **Long** variant.
- If the type of the result is a **Long**, **Single**, or **Date** variant that overflows, then the result is a **Double** variant.

Example

```
'This example assigns string and numeric variable values and
'then uses the + operator to concatenate the strings and form
'the sums of numeric variables.
Sub Main()
      i$ = "Concatenation" + " is fun!"
      j% = 120 + 5      'Addition of numeric literals
      k# = j% + 2.7     'Addition of numeric variable
      MsgBox "This concatenation becomes: '" i$ + _
         Str(j%) + Str(k#) & "'"
End Sub
```

See Also & (operator); Operator Precedence (topic).

Platform(s) All.

< (operator)

See Comparison Operators (topic).

<= (operator)

See Comparison Operators (topic).

<> (operator)

See Comparison Operators (topic).

= (statement)

Syntax *variable = expression*

Description Assigns the result of an expression to a variable.

Comments When assigning expressions to variables, internal type conversions are performed automatically between any two numeric quantities. Thus, you can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This occurs when the larger type contains a numeric quantity that cannot be represented by the smaller type. For example, the following code will produce a runtime error:

```
Dim amount As Long
Dim quantity As Integer
amount = 400123      'Assign a value out of range for int.
quantity = amount    'Attempt to assign to Integer.
```

When performing an automatic data conversion, underflow is not an error.

The assignment operator (=) cannot be used to assign objects. Use the **Set** statement instead.

Example

```
Sub Main()
      a$ = "This is a string"
      b% = 100
      c# = 1213.3443
      MsgBox a$ & "," & b% & "," & c#
End Sub
```

See Also Let (statement); Operator Precedence (topic); Set (statement); Expression Evaluation (topic).

Platform(s) All.

= (operator)

See Comparison Operators (topic).

> (operator)

See Comparison Operators (topic).

>= (operator)

See Comparison Operators (topic).

Abs (function)

Syntax `Abs(`*expression*`)`

Description Returns the absolute value of *expression*.

Comments If *expression* is **Null**, then **Null** is returned. **Empty** is treated as 0.

The type of the result is the same as that of *expression*, with the following exceptions:

- If *expression* is an **Integer** that overflows its legal range, then the result is returned as a **Long**. This only occurs with the largest negative **Integer**:

```
Dim a As Variant
Dim i As Integer
i = -32768
a = Abs(i)          'Result is a Long.
i = Abs(i)          'Overflow!
```

- If *expression* is a **Long** that overflows its legal range, then the result is returned as a **Double**. This only occurs with the largest negative **Long**:

```
Dim a As Variant
Dim l As Long
l = -2147483648
a = Abs(l)          'Result is a Double.
l = Abs(l)          'Overflow!
```

- If *expression* is a **Currency** value that overflows its legal range, an overflow error is generated.

Example

```
'This example assigns absolute values to variables of four types
'and displays the result.
Sub Main()
   s1% = Abs(-10.55)
   s2& = Abs(-10.55)
   s3! = Abs(-10.55)
   s4# = Abs(-10.55)
   MsgBox "The absolute values are: " & s1% & "," & s2& & "," _
      & s3! & "," & s4#
End Sub
```

See Also Sgn (function).

Platform(s) All.

ActivateControl (statement)

Syntax `ActivateControl` *control*

Description Sets the focus to the control with the specified name or ID.

Comments The *control* parameter specifies either the name or the ID of the control to be activated, as shown in the following table:

If control is	Then
String	A control by that name is activated. For push buttons, option buttons, or check boxes, the control with this name is activated. For list boxes, combo boxes, and text boxes, the control that immediately follows the text control with this name is activated.
Numeric	A control with this ID is activated. The ID is first converted to an **Integer**.

The **ActivateControl** statement generates a runtime error if the dialog control referenced by *control* cannot be found.

You can use the **ActivateControl** statement to set the focus to a custom control within a dialog box. First, set the focus to the control that immediately precedes the custom control, then simulate a Tab keypress, as in the following example:

```
ActivateControl "Portrait"
DoKeys "{TAB}"
```

Note: The **ActivateControl** statement is used to activate a control in another application's dialog box. Use the **DlgFocus** statement to activate a control in a dynamic dialog box.

Example

```
'This example runs Notepad using Program Manager's Run command.
'It uses the ActivateControl command to switch focus between the
'different controls of the Run dialog box.
Sub Main()
   If AppFind$("Program Manager") = "" Then Exit Sub
   AppActivate "Program Manager"
   Menu "File.Run"
   SendKeys "Notepad"
   ActivateControl "Run minimized"
   SendKeys " "
   ActivateControl "OK"
   SendKeys "{Enter}"
End Sub
```

See Also DlgFocus (function).

Platform(s) Windows.

And (operator)

Syntax *result* = *expression1* And *expression2*

Description Performs a logical or binary conjunction on two expressions.

Comments If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical conjunction is performed as follows:

If *expression1* **is**	**and** *expression2* **is**	**then the** *result* **is**
True	**True**	**True**
True	**False**	**False**
True	**Null**	**Null**
False	**True**	**False**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**Null**
Null	**False**	**False**
Null	**Null**	**Null**

Binary Conjunction

If the two expressions are **Integer**, then a binary conjunction is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long**, and a binary conjunction is then performed, returning a **Long** result.

Binary conjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in *expression1* **is**	**and bit in** *expression2* **is**	**the** *result* **is**
1	1	1
0	1	0
1	0	0
0	0	0

Examples

```
Sub Main()
   n1 = 1001
   n2 = 1000
   b1 = True
   b2 = False
   'This example performs a numeric bitwise And operation and
   'stores the result in N3.
   n3 = n1 And n2
```

```
   'This example performs a logical And comparing B1 and B2
   'and displays the result.
   If b1 And b2 Then
      MsgBox "b1 and b2 are True; n3 is: " & n3
   Else
      MsgBox "b1 and b2 are False; n3 is: " & n3
   End If
End Sub
```

See Also Operator Precedence (topic); Or (operator); Xor (operator); Eqv (operator); Imp (operator).

Platform(s) All.

AnswerBox (function)

Syntax `AnswerBox(`*prompt* `[,[`*button1*`] [,[`*button2*`] [,[`*button3*`] [,[`*title*`] [,`*helpfile*`,`*context*`]]]]]]])`

Description Displays a dialog box prompting the user for a response and returns an **Integer** indicating which button was clicked (1 for the first button, 2 for the second, and so on).

Comments The **AnswerBox** function takes the following parameters:

Parameter	Description
prompt	Text to be displayed above the text box. The *prompt* parameter can be any expression convertible to a **String**. BasicScript resizes the dialog box to hold the entire contents of *prompt*, up to a maximum width of 5/8 of the width of the screen and a maximum height of 5/8 of the height of the screen. BasicScript word-wraps any lines too long to fit within the dialog box and truncates all lines beyond the maximum number of lines that fit in the dialog box. You can insert a carriage-return/line-feed character in a string to cause a line break in your message. A runtime error is generated if this parameter is **Null**.
button1	The text for the first button. If omitted, then "OK and "Cancel" are used. A runtime error is generated if this parameter is **Null**.
button2	The text for the second button. A runtime error is generated if this parameter is **Null**.
button3	The text for the third button. A runtime error is generated if this parameter is **Null**.

Parameter	Description
title	**String** specifying the title of the dialog. If missing, then the default title is used.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

The width of each button is determined by the width of the widest button.

The **AnswerBox** function returns 0 if the user selects Cancel.

If both the *helpfile* and *context* parameters are specified, then context-sensitive help can be invoked using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Example

```
'This example displays a dialog box containing three buttons. It
'displays an additional message based on which of the three
'buttons is selected.
Sub Main()
   r% = AnswerBox("Copy files?", "Save", "Restore", "Cancel")
   Select Case r%
      Case 1
         MsgBox "Files will be saved."
      Case 2
         MsgBox "Files will be restored."
      Case Else
         MsgBox "Operation canceled."
   End Select
End Sub
```

See Also MsgBox (statement);AskBox, AskBox$ (functions); AskPassword, AskPassword$ (functions); InputBox, InputBox$ (functions); OpenFileName$ (function); SaveFileName$ (function); SelectBox (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Any (data type)

Description Used with the **Declare** statement to indicate that type checking is not to be performed with a given argument.

Comments Given the following declaration:

```
Declare Sub Foo Lib "FOO.DLL" (a As Any)
```

the following calls are valid:

```
Foo 10
Foo "Hello, world."
```

Example

```
'This example calls the FindWindow to determine whether Program
'Manager is running. This example will only run under Windows and
'Win32 platforms.
'This example uses the Any keyword to pass a NULL pointer, which
'is accepted by the FindWindow function.
Declare Function FindWindow16 Lib "user" Alias _
   "FindWindow" (ByVal Class As Any,ByVal Title As Any) As
Integer
Declare Function FindWindow32 Lib "user32" Alias _
"FindWindowA" (ByVal Class As Any,ByVal Title As Any) As Long
Sub Main()
   Dim hWnd As Variant
   If Basic.Os = ebWin16 Then
      hWnd = FindWindow16("PROGMAN",0&)
   ElseIf Basic.Os = ebWin32 Then
      hWnd = FindWindow32("PROGMAN",0&)
   Else
      hWnd = 0
   End If
   If hWnd <> 0 Then
      MsgBox "Program Manager is running, window handle is " _
         & hWnd
   End If
End Sub
```

See Also Declare (statement).

Platform(s) All.

AppActivate (statement)

Syntax `AppActivate` *title* | *taskID*, [*wait*]

Description Activates an application given its name or task ID.

Comments The **AppActivate** statement takes the following named parameters:

Named Parameter	Description
title	A **String** containing the name of the application to be activated.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the **Shell** function.

Named Parameter	Description
wait	An optional boolean value indicating whether BasicScript will wait for calling application to be activated before activating the specified application. If **False** (the default), then BasicScript will activate the specified application immediately.

Note: When activating applications using the task ID, it is important to declare the variable used to hold the task ID as a **Variant**. The type of the ID depends on the platform on which BasicScript is running.

On some platforms, applications don't activate immediately. To compensate, the **AppActivate** statement will wait a maximum of 10 seconds before failing, giving the activated application plenty of time to become activated.

Examples

```
'This example activates Program Manager.
Sub Main()
   AppActivate "Program Manager"
End Sub
'This example runs another application, then activates it.
Sub Main()
   Dim id as variant
   id = Shell("Notepad",7)'Run Notepad minimized.
   AppActivate "Program Manager"'Activate Program Manager.
   AppActivate id       'Now activate Notepad.
End Sub
```

See Also Shell (function);SendKeys (statement);WinActivate (statement).

Platform(s) Windows, Macintosh, Win32, OS/2.

Platform Notes **Windows, Win32:** The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Minimized applications are not restored before activation. Thus, activating a minimized DOS application will not restore it; rather, it will highlight its icon.

A runtime error results if the window being activated is not enabled, as is the case if that application is currently displaying a modal dialog box.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

Macintosh: On the Macintosh, the *title* parameter specifies the title of the desired application. The **MacID** function can be used to specify the application signature of the application to be activated:

```
AppActivate MacID(text$) | task
```

The *title* parameter is a four-character string containing an application signature. A runtime error occurs if the **MacID** function is used on platforms other than the Macintosh.

AppClose (statement)

Syntax `AppClose [`*title* `|` *taskID*`]`

Description Closes the named application.

Comments The *title* parameter is a **String** containing the name of the application. If the *title* parameter is absent, then the **AppClose** statement closes the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example

```
'This example activates Excel, then closes it.
Sub Main()
      If AppFind$("Microsoft Excel") = "" Then
         MsgBox "Excel is not running."
         Exit Sub
      End If
      AppActivate "Microsoft Excel"
      AppClose "Microsoft Excel"
End Sub
```

See Also AppMaximize (statement); AppMinimize (statement); AppRestore (statement); AppMove (statement); AppSize (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** A runtime error results if the application being closed is not enabled, as is the case if that application is currently displaying a modal dialog box.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppFilename$ (function)

Syntax `AppFilename$([`*title* | *taskID*`])`

Description Returns the filename of the named application.

Comments The *title* parameter is a **String** containing the name of the desired application. If the *title* parameter is omitted, then the **AppFilename$** function returns the filename of the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example

```
'This example switches the focus to Excel, then changes the
'current directory to be the same as that of Excel.
Sub Main()
   If AppFind$("Microsoft Excel") = "" Then
      MsgBox "Excel is not running."
      Exit Sub
   End If
   AppActivate "Microsoft Excel"'Activate Excel.
   s$ = AppFilename$'Find where the Excel executable is.
   d$ = FileParse$(s$,2)'Get the path portion of the filename.
   MsgBox d$            'Display directory name.
End Sub
```

See Also AppFind, AppFind$ (functions).

Platform(s) Windows, OS/2.

Platform Notes **Windows, Win32:** For DOS applications launched from Windows, the **AppFilename** function returns the name of the DOS program, not winoldap.exe.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppFind, AppFind$ (functions)

Syntax `AppFind[$] (`*title* | *taskID*`)`

Description Returns a **String** containing the full name of the application matching either *title* or *taskID*.

Comments The *title* parameter specifies the title of the application to find. If there is no exact match, BasicScript will find an application whose title begins with *title*.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

The **AppFind$** functions returns a **String**, whereas the **AppFind** function returns a **String** variant. If the specified application cannot be found, then **AppFind$** returns a zero-length string and **AppFind** returns **Empty**. Using **AppFind** allows you detect failure when attempting to find an application with no caption (i.e., **Empty** is returned instead of a zero-length **String**).

AppFind$ is generally used to determine whether a given application is running. The following expression returns True if Microsoft Word is running:

```
AppFind$("Microsoft Word")
```

Example

```
'This example checks to see whether Excel is running before
'activating it.
Sub Main()
   If AppFind$("Microsoft Excel") <> "" Then
      AppActivate "Microsoft Excel"
   Else
      MsgBox "Excel is not running."
   End If
End Sub
```

See Also AppFilename$ (function).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, this function returns a **String** containing the exact text appearing in the title bar of the active application's main window.

AppGetActive$ (function)

Syntax

```
AppGetActive$()
```

Description Returns a **String** containing the name of the application.

Comments If no application is active, the **AppGetActive$** function returns a zero-length string.

You can use **AppGetActive$** to retrieve the name of the active application. You can then use this name in calls to routines that require an application name.

Example

```
Sub Main()
   n$ = AppGetActive$()
   AppMinimize n$
End Sub
```

See Also AppActivate (statement); WinFind (function).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, this function returns a **String** containing the exact text appearing in the title bar of the active application's main window.

AppGetPosition (statement)

Syntax `AppGetPosition` *x*`,`*y*`,`*width*`,`*height* `[,`*title* `|` *taskID*`]`

Description Retrieves the position of the named application.

Comments The **AppGetPosition** statement takes the following parameters:

Parameter	Description
x, *y*	Names of **Integer** variables to receive the position of the application's window.
width, *height*	Names of **Integer** variables to receive the size of the application's window.
title	A string containing the name of the application. If the *title* parameter is omitted, then the active application is used.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the **Shell** function.

The *x*, *y*, *width*, and *height* variables are filled with the position and size of the application's window. If an argument is not a variable, then the argument is ignored, as in the following example, which only retrieves the *x* and *y* parameters and ignores the *width* and *height* parameters:

```
Dim x as integer, y as integer
AppGetPosition x,y,0,0,"Program Manager"
```

Example

```
Sub Main()
   Dim x As Integer, y As Integer
   Dim cx As Integer, cy As Integer
   AppGetPosition x,y,cx,cy,"Program Manager"
End Sub
```

See Also AppMove (statement); AppSize (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** The position and size of the window are returned in twips.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppGetState (function)

Syntax `AppGetState[([`*title* | *taskID*`])]`

Description Returns an **Integer** specifying the state of the specified top-level window.

Comments The **AppGetState** function returns any of the following values:

If the window is	Then AppGetState returns	Value
Maximized	**ebMinimized**	**1**
Minimized	**ebMaximized**	**2**
Restored	**ebRestored**	**3**

The *title* parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppGetState** function returns the name of the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example

```
'This example saves the state of Program Manager, changes it,
'then restores it to its original setting.
Sub Main()
   If AppFind$("Program Manager") = "" Then
      MsgBox "Can't find Program Manager."
      Exit Sub
   End If
   AppActivate "Program Manager"'Activate Program Manager.
   state = AppGetState'Save its state.
   AppMinimize             'Minimize it.
   MsgBox "Program Manager is minimized. Select OK to restore
it."
   AppActivate "Program Manager"
   AppSetState state    'Restore it.
End Sub
```

See Also AppMaximize (statement); AppMinimize (statement); AppRestore (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows, the *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppHide (statement)

Syntax `AppHide [`*title* `|` *taskID*`]`

Description Hides the named application.

Comments If the named application is already hidden, the **AppHide** statement will have no effect.

The *title* parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppHide** statement hides the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

AppHide generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog box.

Example

```
'This example hides Program Manager.
Sub Main()
   'See whether Program Manager is running.
   If AppFind$("Program Manager") = "" Then Exit Sub
   AppHide "Program Manager"
   MsgBox "Program Manager is hidden. Press OK to show it again."
   AppShow "Program Manager"
End Sub
```

See Also AppShow (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows, the *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppList (statement)

Syntax `AppList` *`AppNames$`*`()`

Description Fills an array with the names of all open applications.

Comments The *AppNames$* parameter must specify either a zero- or one-dimensioned dynamic **String** array or a one-dimensional fixed **String** array. If the array is dynamic, then it will be redimensioned to match the number of open applications. For fixed arrays, **AppList** first erases each array element, then begins assigning application names to the elements in the array. If there are fewer elements than will fit in the array, then the remaining elements are unused. BasicScript returns a runtime error if the array is too small to hold the new elements.

After calling this function, you can use **LBound** and **UBound** to determine the new size of the array.

Example

```
'This example minimizes all applications on the desktop.
Sub Main()
   Dim apps$()
   AppList apps
   'Check to see whether any applications were found.
   If ArrayDims(apps) = 0 Then Exit Sub
   For i = LBound(apps) To UBound(apps)
      AppMinimize apps(i)
   Next i
End Sub
```

See Also WinList (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the name of an application is considered to be the exact text that appears in the title bar of the application's main window.

AppMaximize (statement)

Syntax `AppMaximize [`*`title`* `|` *`taskID`*`]`

Description Maximizes the named application.

Comments The *title* parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppMaximize** function maximizes the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example
```
Sub Main()
   AppMaximize "Program Manager"'Maximize Program Manager.
   If AppFind$("NotePad") <> "" Then
      AppActivate "NotePad"'Set the focus to NotePad.
      AppMaximize       'Maximize it.
   End If
End Sub
```

See Also AppMinimize (statement); AppRestore (statement); AppMove (statement); AppSize (statement); AppClose (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** If the named application is maximized or hidden, the **AppMaximize** statement will have no effect.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppMaximize generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog box.

AppMinimize (statement)

Syntax `AppMinimize [`*title* `|` *taskID*`]`

Description Minimizes the named application.

Comments The *title* parameter is a **String** containing the name of the desired application. If it is omitted, then the **AppMinimize** function minimizes the active application.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example
```
Sub Main()
   AppMinimize "Program Manager"'Maximize Program Manager.
   If AppFind$("NotePad") <> "" Then
      AppActivate "NotePad"'Set the focus to NotePad.
```

```
      AppMinimize       'Maximize it.
   End If
End Sub
```

See Also AppMaximize (statement); AppRestore (statement); AppMove (statement); AppSize (statement); AppClose (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** If the named application is minimized or hidden, the **AppMinimize** statement will have no effect.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppMinimize generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog box.

AppMove (statement)

Syntax `AppMove x,y [,title | taskID]`

Description Sets the upper left corner of the named application to a given location.

Comments The **AppMove** statement takes the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the upper left corner of the new location of the application, relative to the upper left corner of the display.
title	**String** containing the name of the application to move. If this parameter is omitted, then the active application is moved.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the **Shell** function.

Example

```
'This example activates Program Manager, then moves it 10 pixels
'to the right.
Sub Main()
   Dim x%,y%
```

```
   AppActivate "Program Manager"'Activate Program Manager.
   AppGetPosition x%,y%,0,0'Retrieve its position.
   x% = x% + Screen.TwipsPerPixelX * 10'Add 10 pixels.
   AppMove x% + 10,y%'Nudge it 10 pixels to the right.
End Sub
```

See Also AppMaximize (statement); AppMinimize (statement); AppRestore (statement); AppSize (statement); AppClose (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** If the named application is maximized or hidden, the **AppMove** statement will have no effect.

The *x* and *y* parameters are specified in twips.

AppMove will accept *x* and *y* parameters that are off the screen.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppMove generates a runtime error if the named application is not enabled, as is the case if that application is currently displaying a modal dialog box.

AppRestore (statement)

Syntax `AppRestore [`*title* `|` *taskID*`]`

Description Restores the named application.

Comments The *title* parameter is a **String** containing the name of the application to restore. If this parameter is omitted, then the active application is restored.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example

```
'This example minimizes Program Manager, then restores it.
Sub Main()
   If AppFind$("Program Manager") = "" Then Exit Sub
   AppActivate "Program Manager"
   AppMinimize "Program Manager"
   MsgBox "Program Manager is minimized. Press OK to restore it."
   AppRestore "Program Manager"
```

```
End Sub
```

See Also AppMaximize (statement); AppMinimize (statement); AppMove (statement); AppSize (statement); AppClose (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows, the *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title,* then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppRestore will have an effect only if the main window of the named application is either maximized or minimized.

AppRestore will have no effect if the named window is hidden.

AppRestore generates a runtime error if the named application is not enabled, as is the case if that application is currently displaying a modal dialog box.

AppSetState (statement)

Syntax `AppSetState` *newstate* `[,`*title* `|` *taskID*`]`

Description Maximizes, minimizes, or restores the named application, depending on the value of *newstate*.

Comments The **AppSetState** statement takes the following parameters:

Parameter	Description
newstate	An **Integer** specifying the new state of the window.
title	A **String** containing the name of the application to change. If omitted, then the active application is used.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the **Shell** function.

The *newstate* parameter can be any of the following values:

Constant	Value	Description
ebMinimized	**1**	The named application is minimized.

Constant	Value	Description
ebMaximized	**2**	The named application is maximized.
ebRestored	**3**	The named application is restored.

Example See **AppGetState** (function).

See Also AppGetState (function); AppMinimize (statement); AppMaximize (statement); AppRestore (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows, the *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppShow (statement)

Syntax `AppShow [`*title* `|` *taskID*`]`

Description Makes the named application visible.

Comments The *title* parameter is a **String** containing the name of the application to show. If this parameter is omitted, then the active application is shown.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example See **AppHide** (statement).

See Also AppHide (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** If the named application is already visible, **AppShow** will have no effect.

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

AppShow generates a runtime error if the named application is not enabled, as is the case if that application is displaying a modal dialog box.

AppSize (statement)

Syntax `AppSize` *width*`,`*height* `[,`*title* `|` *taskID*`]`

Description Sets the width and height of the named application.

Comments The **AppSize** statement takes the following parameters:

Parameter	Description
width, height	**Integer** coordinates specifying the new size of the application.
title	**String** containing the name of the application to resize. If this parameter is omitted, then the active application is use.
taskID	A number specifying the task ID of the application to be activated. Acceptable task IDs are returned by the **Shell** function.

Example

```
'This example enlarges the active application by 10 pixels in
'both the vertical and horizontal directions.
Sub Main()
   Dim w%,h%
   AppGetPosition 0,0,w%,h%'Get current width/height.
   x% = x% + Screen.TwipsPerPixelX * 10'Add 10 pixels.
   y% = y% + Screen.TwipsPerPixelY * 10'Add 10 pixels.
   AppSize w%,h%           'Change to new size.
End Sub
```

See Also AppMaximize (statement); AppMinimize (statement); AppRestore (statement); AppMove (statement); AppClose (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** The *width* and *height* parameters are specified in twips.

This statement will only work if the named application is restored (i.e., not minimized or maximized).

The *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

A runtime error results if the application being resized is not enabled, which is the case if that application is displaying a modal dialog box when an **AppSize** statement is executed.

AppType (function)

Syntax `AppType [(`*title* | *taskID*`)]`

Description Returns an **Integer** indicating the executable file type of the named application:

Returns	If the file type is:
ebDos	DOS executable
ebWindows	Windows executable

Comments The *title* parameter is a **String** containing the name of the application. If this parameter is omitted, then the active application is used.

Alternatively, you can specify the ID of the task as returned by the **Shell** function.

Example

```
'This example creates an array of strings containing the names
'of all the running Windows applications. It uses the AppType
'command to determine whether an application is a Windows
'application or a DOS application.
Sub Main()
   Dim apps$(),wapps$()
   AppList apps'Retrieve a list of all Windows and DOS apps.
   If ArrayDims(apps) = 0 Then
      MsgBox "There are no running applications."
      Exit Sub
   End If
   'Create an array to hold only the Windows apps.
   ReDim wapps$(UBound(apps))
   n = 0'Copy the Windows apps from one array to the target array.
   For i = LBound(apps) to UBound(apps)
      If AppType(apps(i)) = ebWindows Then
         wapps(n) = apps(i)
```

```
            n = n + 1
         End If
      Next i
      If n = 0 Then'Make sure at least one Windows app was found.
         MsgBox "There are no running Windows applications."
         Exit Sub
      End If
      ReDim Preserve wapps(n - 1) 'Resize to hold the exact number.
      'Let the user pick one.
      index% = SelectBox("Apps","Select an application:",wapps)
End Sub
```

See Also AppFilename$ (function).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows, the *title* parameter is the exact string appearing in the title bar of the named application's main window. If no application is found whose title exactly matches *title*, then a second search is performed for applications whose title string begins with *title*. If more than one application is found that matches *title*, then the first application encountered is used.

Under Windows 95, applications adhere to a convention where the caption contains the name of the file before the name of the application. For example, under NT, the caption for Notepad is "Notepad - (Untitled)", whereas under Windows 95, the caption is "Untitled - Notepad". You must keep this in mind when specifying the *title* parameter.

ArrayDims (function)

Syntax `ArrayDims(`*arrayvariable*`)`

Description Returns an **Integer** containing the number of dimensions of a given array.

Comments This function can be used to determine whether a given array contains any elements or if the array is initially created with no dimensions and then redimensioned by another function, such as the **FileList** function, as shown in the following example.

Example

```
'This example allocates an empty (null-dimensioned) array; fills
'the array with a list of filenames, which resizes the array;
'then tests the array dimension and displays an appropriate
'message.
Sub Main()
   Dim f$()
   FileList f$,"c:\*.bat"
   If ArrayDims(f$) = 0 Then
      MsgBox "The array is empty."
   Else
      MsgBox "The array size is: " & (UBound(f$) - UBound(f$) + 1)
```

```
   End If
End Sub
```

See Also LBound (function);UBound (function); Arrays (topic).

Platform(s) All.

Arrays (topic)

Declaring Array Variables

Arrays in BasicScript are declared using any of the following statements:

```
Dim
Public
Private
```

For example:

```
Dim a(10) As Integer
Public LastNames(1 to 5,-2 to 7) As Variant
Private
```

Arrays of any data type can be created, including **Integer**, **Long**, **Single**, **Double**, **Boolean**, **Date**, **Variant**, **Object**, user-defined structures, and data objects.

The lower and upper bounds of each array dimension must be within the following range:

```
-32768 <= bound <= 32767
```

Arrays can have up to 60 dimensions.

Arrays can be declared as either fixed or dynamic, as described below.

Fixed Arrays

The dimensions of fixed arrays cannot be adjusted at execution time. Once declared, a fixed array will always require the same amount of storage. Fixed arrays can be declared with the **Dim**, **Private**, or **Public** statement by supplying explicit dimensions. The following example declares a fixed array of eleven strings (assuming the option base is 0):

```
Dim a(10) As String
```

Fixed arrays can be used as members of user-defined data types. The following example shows a structure containing fixed-length arrays:

```
Type Foo
   rect(4) As Integer
   colors(10) As Integer
End Type
```

Only fixed arrays can appear within structures.

Dynamic Arrays

Dynamic arrays are declared without explicit dimensions, as shown below:

```
Public Ages() As Integer
```

Dynamic arrays can be resized at execution time using the **Redim** statement:

```
Redim Ages$(100)
```

Subsequent to their initial declaration, dynamic arrays can be redimensioned any number of times. When redimensioning an array, the old array is first erased unless you use the **Preserve** keyword, as shown below:

```
Redim Preserve Ages$(100)
```

Dynamic arrays cannot be members of user-defined data types.

Passing Arrays

Arrays are always passed by reference. When you pass an array, you can specify the array name by itself, or with parentheses as shown below:

```
Dim a(10) As String
FileList a          'Both of these are OK
FileList a()
```

Querying Arrays

The following table describes the functions used to retrieve information about arrays.

Use this function	To
LBound	Retrieve the lower bound of an array. A runtime is generated if the array has no dimensions.
UBound	Retrieve the upper bond of an array. A runtime error is generated if the array has no dimensions.
ArrayDims	Retrieve the number of dimensions of an array. This function returns 0 if the array has no dimensions.

Operations on Arrays

The following table describes the function that operate on arrays:

Use the command	To
ArraySort	Sort an array of integers, longs, singles, doubles, currency, `Boolean`s, dates, or variants.
FileList	Fill an array with a list of files in a given directory.
DiskDrives	Fill an array with a list of valid drive letters.
AppList	Fill an array with a list of running applications.

Use the command	To
WinList	Fill an array with a list of top-level windows.
SelectBox	Display the contents of an array in a list box.
PopupMenu	Display the contents of an array in a popup menu.
ReadInSection	Fill an array with the item names from a section in an INI file.
FileDirs	Fill an array with a list of subdirectories.
Erase	Erase all the elements of an array.
ReDim	Establish the bounds and dimensions of an array.
Dim	Declare an array.

ArraySort (statement)

Syntax `ArraySort` *array*`()`

Description Sorts a single-dimensioned array in ascending order.

Comments If a string array is specified, then the routine sorts alphabetically in ascending order using case-sensitive string comparisons. If a numeric array is specified, the **ArraySort** statement sorts smaller numbers to the lowest array index locations.

BasicScript generates a runtime error if you specify an array with more than one dimension.

When sorting an array of variants, the following rules apply:

- A runtime error is generated if any element of the array is an object.
- **String** is greater than any numeric type.
- **Null** is less than **String** and all numeric types.
- **Empty** is treated as a number with the value 0.
- **String** comparison is case-sensitive (this function is not affected by the **Option Compare** setting).

Example

```
'This example dimensions an array and fills it with filenames
'using FileList, then sorts the array and displays it in a
'select box.
Sub Main()
   Dim f$()
   FileList f$,"c:\*.*"
   ArraySort f$
   r% = SelectBox("Files","Choose one:",f$)
```

```
End Sub
```

See Also ArrayDims (function); LBound (function);UBound (function).

Platform(s) All.

Asc, AscB, AscW(functions)

Syntax

```
Asc(string)
AscB(string)
AscW(string)
```

Description Returns an **Integer** containing the numeric code for the first character of *string*.

Comments This function returns the character value of the first character of *string*. On single-byte systems, this function returns a number between 0 and 255, whereas on MBCS systems, this function returns a number between -32768 and 32767. On wide platforms, this function returns the MBCS character code after converting the wide character to MBCS.

To return the value of the first byte of a string, use the **AscB** function. This function is used when you need the value of the first byte of a string known to contain byte data rather than character data. On single-byte systems, the **AscB** function is identical to the **Asc** function.

On platforms where BasicScript uses wide string internally (such as Win32), the AscW function returns the character value native to that platform. For example, on Win32 platforms, this function returns the UNICODE character code. On single-byte and MBCS platforms, the **AscW** function is equivalent to the **Asc** function.

The following table summarizes the values returned by these functions:

Function	String Format	Returns
Asc		Value of the first byte of *string* (between 0 and 255)
	MBCS	Value of the first character of *string* (between -32769 and 32767)
	Wide	Value of the first character of *string* after conversion to MBCS.
AscB		Value of the first byte of *string*.
	MBCS	Value of the first byte of *string*.
	Wide	Value of the first byte of *string*.
AscW		Same as **Asc**.
	MBCS	Same as **Asc**.
	Wide	Value of the wide character native to the operating system.

Example

```
'This example fills an array with the ASCII values of the
'string's components and displays the result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   s$ = InputBox("Please enter a string.","Enter String")
   If s$ = "" Then End'Exit if no string entered.
   For i = 1 To Len(s$)
      message = message & Asc(Mid$(s$,i,1)) & crlf
   Next i
   MsgBox "The Asc values of the string are:" & message
End Sub
```

See Also Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ (functions).

Platform(s) All.

AskBox, AskBox$ (functions)

Syntax `AskBox[$](`*prompt$* `[,[`*default*`$] [,[`*title*`$][,`*helpfile*`,`*context*`]]])`

Description Displays a dialog box requesting input from the user and returns that input as a **String**.

Comments The **AskBox/AskBox$** functions take the following parameters:

Parameter	Description
prompt$	**String** containing the text to be displayed above the text box. The dialog box is sized to the appropriate width depending on the width of *prompt$*. A runtime error is generated if *prompt$* is **Null**.
default$	**String** containing the initial content of the text box. The user can return the default by immediately selecting OK. A runtime error is generated if *default$* is **Null**.
title$	**String** specifying the title of the dialog. If missing, then the default title is used.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

The **AskBox$** function returns a **String** containing the input typed by the user in the text box. A zero-length string is returned if the user selects Cancel.

The **AskBox** function returns a **String** variant containing the input typed by the user in the text box. An **Empty** variant is returned if the user selects Cancel.

When the dialog box is displayed, the text box has the focus.

The user can type a maximum of 255 characters into the text box displayed by **AskBox$**.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Example

```
'This example asks the user to enter a filename and then
'displays what he or she has typed.
Sub Main()
   s$ = AskBox$("Type in the filename:")
   MsgBox "The filename was: " & s$
End Sub
```

See Also MsgBox (statement); AskPassword, AskPassword$ (functions); InputBox, InputBox$ (functions); OpenFileName$ (function); SaveFileName$ (function); SelectBox (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

AskPassword, AskPassword$ (functions)

Syntax `AskPassword[$](`*prompt$* `[,[`*title$*`] [,`*helpfile*`,`*context*`]])`

Description Returns a **String** containing the text that the user typed.

Comments Unlike the **AskBox/AskBox$** functions, the user sees asterisks in place of the characters that are actually typed. This allows the hidden input of passwords.

The **AskPassword/AskPassword$** functions take the following parameters:

Parameter	Description
prompt$	**String** containing the text to be displayed above the text box. The dialog box is sized to the appropriate width depending on the width of *prompt$*. A runtime error is generated if *prompt$* is **Null**.
title$	**String** specifying the title of the dialog. If missing, then the default title is used.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.

Parameter	Description
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

When the dialog box is first displayed, the text box has the focus.

A maximum of 255 characters can be typed into the text box.

The **AskPassword$** function returns the text typed into the text box, up to a maximum of 255 characters. A zero-length string is returned if the user selects Cancel.

The **AskPassword** function returns a **String** variant. An **Empty** variant is returned if the user selects Cancel.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Example

```
Sub Main()
   s$ = AskPassword$("Type in the password:")
   MsgBox "The password entered is: " & s$
End Sub
```

See Also MsgBox (statement); AskBox, AskBox$ (functions); InputBox, InputBox$ (functions); OpenFileName$ (function); SaveFileName$ (function); SelectBox (function); AnswerBox (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Atn (function)

Syntax `Atn(`*number*`)`

Description Returns the angle (in radians) whose tangent is *number*.

Comments Some helpful conversions:

- Pi (3.1415926536) radians = 180 degrees.
- 1 radian = 57.2957795131 degrees.
- 1 degree = .0174532925 radians.

Example

```
'This example finds the angle whose tangent is 1 (45 degrees)
'and displays the result.
Sub Main()
   a# = Atn(1.00)
```

```
   MsgBox "1.00 is the tangent of " & a# _
      & " radians (45 degrees)."
End Sub
```

See Also Tan (function); Sln (function); Cos (function).

Platform(s) All.

Basic.Architecture$ (property)

Syntax `Basic.Architecture$`

Description Returns a **String** containing the CPU architecture on which BasicScript is executing.

Comments The following table describes what **Basic.Architecture$** returns on various platforms:

Platform	Sample return Value from Basic.Architecture$
Windows	**"Intel"**
Win32	**"Intel"**, **"MIPS"**, **"Alpha AXP"**, or **"PowerPC"**
OS/2	**"Intel"**
NetWare	**"Intel", "Motorola"**
Macintosh	**"PowerPC", "68K"**
UNIX	"i386", "i486"

The **Basic.Architecture$** property returns an empty string if the architecture cannot be determined by BasicScript.

Example

```
'
'Print the CPU architecture...
'
Sub Main()
   MsgBox Basic.Architecture$
End Sub
```

See Also Basic.Processor$ (property); Basic.ProcessorCount (property).

Platform(s) All.

Basic.Capability (method)

Syntax `Basic.Capability(`*which*`)`

Description Returns **True** if the specified capability exists on the current platform; returns **False** otherwise.

Comments The *which* parameter is an **Integer** specifying the capability for which to test. It can be any of the following values:

Value	Returns True If
1	The platform supports disk drives
2	The platform supports system file attribute (**ebSystem**)
3	The platform supports the hidden file attribute (**ebHidden**)
4	The platform supports the volume label file attribute (**ebVolume**)
5	The platform supports the archive file attribute (**ebArchive**)
6	The platform supports denormalized floating-point math
7	The platform supports file locking (i.e., the **Lock** and **Unlock** statements)
8	The platform uses big endian byte ordering
9	The internal string format used by BasicScript uses 2-byte characters.
10	The internal string format used by BasicScript is MBCS.
11	The platform supports wide characters.
12	The platform is MBCS.

Example

```
'This example tests to see whether your current platform
'supports disk drives and hidden file attributes and displays
'the result.
Sub Main()
   message = "This operating system "
   If Basic.Capability(1) Then
      message = message & "supports disk drives."
   Else
      message = message & "does not support disk drives."
   End If
   MsgBox message
End Sub
```

See Also Cross-Platform Scripting (topic);Basic.OS (property).

Platform(s) All.

Basic.CodePage (property)

Syntax `Basic.CodePage`

Description Returns an **Integer** representing the code page for the current locale.

Comments Under Windows, Win32, NetWare, and OS/2, this property returns ANSI code page for the current locale, such as 437 for MS-DOS Latin US or 932 for Japanese.

On the Macintosh, this property returns a number from 0 to 32 containing the script code (e.g., 0 for Roman, 1 for Japanese, and so on) as defined by Apple.

Example

```
Sub Main
   If Basic.OS = ebWin16 And Basic.CodePage = 437 Then
      MsgBox "Running US Windows"
   Else if Basic.OS = ebWin32 And Basic.CodePage = 932 Then
      MsgBox "Japanese NT"
   End If
End Sub
```

See Also Basic.Locale$ (property); Basic.OS (property).

Platform(s) All.

Basic.Eoln$ (property)

Syntax

```
Basic.Eoln$
```

Description Returns a **String** containing the end-of-line character sequence appropriate to the current platform.

Comments This string will be either a carriage return, a carriage return/line feed, or a line feed.

Example

```
'This example writes two lines of text in a message box.
Sub Main()
   MsgBox "This is the first line of text." & Basic.Eoln$ _
      & "This is the second line of text."
End Sub
```

See Also Cross-Platform Scripting (topic); Basic.PathSeparator$ (property).

Platform(s) All.

Basic.FreeMemory (property)

Syntax

```
Basic.FreeMemory
```

Description Returns a **Long** representing the number of bytes of free memory in BasicScript's data space.

Comments This function returns the size of the largest free block in BasicScript's data space. Before this number is returned, the data space is compacted, consolidating free space into a single contiguous free block.

BasicScript's data space contains strings and dynamic arrays.

Example

```
'This example displays free memory in a dialog box.
Sub Main()
   MsgBox "The largest free memory block is: " & Basic.FreeMemory
End Sub
```

See Also System.TotalMemory (property); System.FreeMemory (property); System.FreeResources (property); Basic.FreeMemory (property).

Platform(s) All.

Basic.HomeDir$ (property)

Syntax `Basic.HomeDir$`

Description Returns a **String** specifying the directory containing BasicScript.

Comments This method is used to find the directory in which the BasicScript files are located.

Example

```
'This example assigns the home directory to HD and displays it.
Sub Main()
   hd$ = Basic.HomeDir$
   MsgBox "The BasicScript home directory is: " & hd$
End Sub
```

See Also System.WindowsDirectory$ (property).

Platform(s) All.

Basic.Locale$ (property)

Syntax `Basic.Locale$`

Description Returns a **String** containing the locale under which BasicScript is running.

Comments The locale helps you identify information about your environment, such as the date formats, time format, and other country-sensitive information.

The following table describes the returned value from **Basic.Locale$** on various platforms:

Platform	Return value from Basic.Locale$
Win32	Returns a string in the format: *abbrevlang,langid,nativelang,englang* *abbrevlang*: Three-letter name of the language. This name is formed by taking the two-letter language abbreviation as found in the ISO Standard 639 and adding a third letter, as appropriate, to indicate the sublanguage. This is the same as that name found in the **sLanguage** item in the intl section of the Windows 3.1 WIN.INI file. *langid*: Language ID as defined by the operating system. *nativelang*: Native name of the language. *englang*: Full english name of the language as defined by ISO standard 639.
Windows	Returns a string in the format: *abbrevlang,country* *country*: Native name of the country. *abbrevlang*: Three-letter name of the language. This name is formed by taking the two-letter language abbreviation as found in the ISO Standard 639 and adding a third letter, as appropriate, to indicate the sublanguage. This is the same as that name found in the **sLanguage** item in the intl section of the Windows 3.1 WIN.INI file.
Netware	Returns a string in the following format: *countrycode* [,*countryname*] *countrycode*: Country code based on the telephone country code (1 = US, 2 = Canada, and so on). *countryname*: Name of the country (such as "USA"). The name of country is only provided for NetWare version 4.0 or later.

Platform	Return value from Basic.Locale$
OS/2	Returns a string in the following format: *countrycode*,[*localename*] The parameters are defined as follows: *countrycode*: Country code based on the telephone country code (with the exception of Canada, which uses 2). *localename*: Name of the locale as identified by the **LC_ALL** or **LANG** environment variables. If this parameter is missing, then the host application is using the default C language locale.
UNIX	???
Macintosh	Returns a string in the following format: *langcode*,*langname* *langcode*: A number representing the current language (e.g., 0 for English, 1 for French, 11 for Japanese, and so on). *langname*: The English language name of the language.

Example

```
'
'This example checks to see if we are running in a Japanese
'version of Windows.
'
Sub Main
   If Basic.OS = ebWin16 And Item$(Basic.Locale$,1) = "jpn" Then
      MsgBox "Running Windows on a Japanese computer."
   End If
End Sub
```

See Also Basic.OS (property); Basic.CodePage (property).

Platform(s) All.

Basic.OperatingSystem$ (property)

Syntax `Basic.OperatingSystem$`

Description Returns a **String** containing the name of the operating system.

Comments The following table describes the values returned by this function:

Platform	Sample values returned by Basic.OperatingSystem$
Windows	**"Windows"**, **"Windows for Workgroups"**

Platform	Sample values returned by Basic.OperatingSystem$
Win32	**"Win32s"**, **"Windows 95"**, **"Windows NT"**
OS/2	**"OS/2"**
Macintosh	**"Macintosh"**
Netware	**"NetWare"**
UNIX	"Linux", "sco", "UNIX_SV"

The version of the operating system is determined by calling Basic.**OperatingSystemVersion$**.

Example

```
'
'This script checks the Windows version for special networking
’capabilities.
'
Sub Main()
   If Basic.OS = ebWin16 Then
      If Basic.OperatingSystem$ = "Windows" Then
      MsgBox "Special networking capabilities aren’t present."
   ElseIf Basic.OperatingSystem$ = "Windows for Workgroups" Then
      MsgBox "Network capabilities are present."
   End If
End Sub
```

See Also Basic.OperatingSystemVendor$ (property); Basic.OperatingSystemVersion$ (property); Basic.OS (property).

Platform(s) All.

Basic.OperatingSystemVendor$ (property)

Syntax `Basic.OperatingSystemVendor$`

Description Returns a **String** containing the version of the operating system under which BasicScript is running.

Comments The following table describes the what this function returns for various platforms:

Platform	Sample return value from Basic.OperatingSystemVendor$
Windows	**"Microsoft"**
Win32	**"Microsoft"**
OS/2	**"IBM"**
Netware	Returns the name of the company that distributed NetWare.

Platform	Sample return value from Basic.OperatingSystemVendor$
Macintosh	**"Apple"**
UNIX	"Novell System Laboratories", "Linux", Santa Cruz Operations"

The name of the operating system is returned by the **Basic.OperatingSystem$** property. The version of the operating system is determined by the **Basic.OperatingSystemVersion$** property.

Example

```
'
'The following example prints the operating system vendor
'
Sub Main
   MsgBox "The manufacturer of the operating system is: " & _
      Basic.OperatingSystemVendor$
End Sub
```

See Also Basic.OperatingSystem$ (property); Basic.OperatingSystemVersion$ (property); Basic.OS (property).

Platform(s) All.

Basic.OperatingSystemVersion$ (property)

Syntax

```
Basic.OperatingSystemVersion$
```

Description Returns a **String** containing the version of the operating system under which BasicScript is running.

Example

```
'
'This example checks the Windows version to ensure that a
'feature is supported.
'
Sub Main
   If Basic.OperatingSystem$ = "Windows"
      If Basic.OperatingSystemVersion$ <= 3 Then
         MsgBox "That feature is not supported."
      Else
         MsgBox "Windows version 3.1 or greater"
      End If
   End If
End Sub
```

See Also Basic.OperatingSystem$ (property);Basic.OperatingSystemVendor$ (property); Basic.OS (property).

Platform(s) All.

Platform Notes **Win32, Macintosh:** The version number is returned in the following format:
major.*minor*.*buildnumber*

The parts of the version number are described in the following table:

Part	Description
major	Identifies the major version number of the operating system.
minor	Identifies the minor version number of the operating system.
buildnumber	Identifies the build number of the operating system.

Windows, NetWare, OS/2: The version number is returns as *major*.*minor*.

UNIX: The version returned does not follow a standard format and is specific to the operating system.

Basic.OS (property)

Syntax `Basic.OS`

Description Returns an **Integer** indicating the current platform.

Comments

Value	Constant	Platform
0	**ebWin16**	Microsoft Windows
2	**edWin32**	Microsoft Windows 95
		Microsoft Windows NT Workstation (Intel, Alpha, AXP, MIPS,)
		Microsoft Windows NT Server (Intel, Alpha, AXP, MIPS)
		Microsoft Win32s running under Windows 3.1
3	**ebSolaris**	Sun Solaris 2.x
4	**ebSunOS**	SunOS
5	**ebHPUX**	HP-UX
7	**ebIrix**	Silicon Graphics IRIX
8	**ebAIX**	IBM AIX
9	**ebNetWare**	Novell NetWare
10	**ebMacintosh**	Apple Macintosh
11	**ebOS2**	IBM OS/2

The value returned is not necessarily the platform under which BasicScript is running but rather an indicator of the platform for which BasicScript was created. For example, it is possible to run BasicScript for Windows under Windows NT Workstation. In this case, **Basic.OS** will return 0.

Example

```
'This example determines the operating system for which this
'version was created and displays the appropriate message.
Sub Main()
   Select Case Basic.OS
      Case ebWin16
         s = "Windows"
      Case ebNetWare
         s = "NetWare"
      Case Else
         s = "neither Windows nor NetWare"
   End Select
   MsgBox "You are currently running " & s
End Sub
```

See Also Cross-Platform Scripting (topic).

Platform(s) All.

Basic.PathSeparator$ (property)

Syntax

```
Basic.PathSeparator$
```

Description Returns a **String** containing the path separator appropriate for the current platform.

Comments The returned string is any one of the following characters: / (slash), \ (back slash), **:** (colon).

Example

```
Sub Main()
   MsgBox "The path separator for this platform is: " _
      & Basic.PathSeparator$
End Sub
```

See Also Basic.Eoln$ (property); Cross-Platform Scripting (topic).

Platform(s) All.

Basic.Processor$ (property)

Syntax

```
Basic.Processor$
```

Description Returns a **String** containing the name of the CPU in the computer on which BasicScript is running.

Comments You can retrieve the number of processors within the computer using the **Basic.ProcessorCount** property.

The following table describes the possible values returned by this property:

Platform	Sample values returned from Basic.Processor$
Windows	**"8086"**, **"80186"**, **"80286"**, **"80386"**, **"80486"**. On Pentium computers, the value **"80486"** is returned.
Win32	On Intel platforms, one of the following is returned: **"80386"**, **"80486"**, **"Pentium"**. On MIPS platforms, the string **"R***x***"** is returned, such as **"R4000"**. On Alpha platforms, one of the following is returned: **"321064"**, **"321066"**, **"321164"**. On PowerPC platforms, one of the following is returned: **"601"**, **"603"**, **"604"**, **"603+"**, **"604+"**, **"620"**.
OS/2	"80386", "80486", "Pentium".
UNIX	"i386", "i486"
NetWare	**"680x0"**, **"80x86"**
Macintosh	On 68K platforms, one of the following is returned: **"68000"**, **"68010"**, **"68020"**, **"68030"**, **"68040"**. On PowerMac platforms, the string **"601"** is returned.

An empty string is returned if BasicScript cannot determine the processor type.

Example

```
'
'This example prints the CPU of the computer on which
'BasicScript is executing.
'
Sub Main()
   MsgBox "Processor = " & Basic.Processor$
End Sub
```

See Also Basic.ProcessorCount (property).

Platform(s) All.

Basic.ProcessorCount (property)

Syntax `Basic.ProcessorCount`

Description Returns the number of CPUs installed on the computer on which BasicScript is running.

Comments You can determine the type of processor using the **Basic.Processor$** property.

This property return 1 if the CPU has only one processor or is otherwise incapable of containing more than one processor.

Example

```
'
'Print the number of processors in the computer.
'
Sub Main()
   MsgBox "There are " & Basic.ProcessorCount & _
      " processor(s) in the computer."
End Sub
```

See Also Basic.Processor$ (property).

Platform(s) All.

Basic.Version$ (property)

Syntax `Basic.Version$`

Description Returns a **String** containing the version of BasicScript.

Comments This function returns the major and minor version numbers in the format *major.minor.BuildNumber*, as in "2.00.30."

Example

```
'This example displays the current version of BasicScript.
Sub Main()
   MsgBox "Version " & Basic.Version$ _
      & " of BasicScript is running"
End Sub
```

Platform(s) All.

Beep (statement)

Syntax `Beep`

Description Makes a single system beep.

Example

```
'This example causes the system to beep five times and displays
'a reminder message.
Sub Main()
   For i = 1 To 5
      Beep
      Sleep(200)
   Next i
   MsgBox "You have an upcoming appointment!"
End Sub
```

See Also Mci (function).

Platform(s) All.

Begin Dialog (statement)

Syntax

```
Begin Dialog DialogName [x],[y],width,height,title$ [,[.DlgProc]
[,[PicName$] [,style]]]
        Dialog Statements
End Dialog
```

Description Defines a dialog box template for use with the **Dialog** statement and function.

Comments A dialog box template is constructed by placing any of the following statements between the **Begin Dialog** and **End Dialog** statements (no other statements besides comments can appear within a dialog box template):

Picture	**PictureButton**	**OptionButton**
OptionGroup	**CancelButton**	**Text**
TextBox	**GroupBox**	**DropListBox**
ListBox	**ComboBox**	**CheckBox**
PushButton	**OKButton**	

The **Begin Dialog** statement requires the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the position of the upper left corner of the dialog box relative to the parent window. These coordinates are in dialog units. If either coordinate is unspecified, then the dialog box will be centered in that direction on the parent window.
width, *height*	**Integer** coordinates specifying the width and height of the dialog box (in dialog units).
DialogName	Name of the dialog box template. Once a dialog box template has been created, a variable can be dimensioned using this name.
title$	**String** containing the name to appear in the title bar of the dialog box. If this parameter specifies a zero-length string, then the name "BasicScript" is used.

Parameter	Description
.DlgProc	Name of the dialog function. The routine specified by *.DlgProc* will be called by BasicScript when certain actions occur during processing of the dialog box. (See **DlgProc** [prototype] for additional information about dialog functions.) If this parameter is omitted, then BasicScript processes the dialog box using the default dialog box processing behavior.
PicName$	**String** specifying the name of a DLL containing pictures. This DLL is used as the origin for pictures when the picture type is 10. If this parameter is omitted, then no picture library will be used.
style	Specifies extra styles for the dialog. It can be any of the following values: 0- Dialog does not contain a title or close box. 1 - Dialog contains a title and no close box. 2 (or omitted) - Dialog contains both the title and close box.

BasicScript generates an error if the dialog box template contains no controls.

A dialog box template must have at least one **PushButton**, **OKButton**, or **CancelButton** statement. Otherwise, there will be no way to close the dialog box.

Dialog units are defined as 1/4 the width of the font in the horizontal direction and 1/8 the height of the font in the vertical direction.

Any number of user dialog boxes can be created, but each one must be created using a different name as the *DialogName.* Only one user dialog box may be invoked at any time.

Expression Evaluation within the Dialog Box Template

The **Begin Dialog** statement creates the template for the dialog box. Any expression or variable name that appears within any of the statements in the dialog box template is not evaluated until a variable is dimensioned of type *DialogName*. The following example shows this behavior:

```
MyTitle$ = "Hello, World"
Begin Dialog MyTemplate 16,32,116,64,MyTitle$
   OKButton 12,40,40,14
End Dialog
MyTitle$ = "Sample Dialog"
Dim Dummy As MyTemplate
rc% = Dialog(Dummy)
```

The above example creates a dialog box with the title "Sample Dialog".

Expressions within dialog box templates cannot reference external subroutines or functions.

All controls within a dialog box use the same font. The fonts used for the text and text box controls can be changed explicitly by setting the font parameters in the **Text** and **TextBox** statements. A maximum of 128 fonts can be used within a single dialog box, although the practical limitation may be less.

Example

```
'This example creates an exit dialog box.
Sub Main()
   Begin Dialog QuitDialogTemplate 16,32,116,64,"Quit"
      Text 4,8,108,8,"Are you sure you want to exit?"
      CheckBox 32,24,63,8,"Save Changes",.SaveChanges
      OKButton 12,40,40,14
      CancelButton 60,40,40,14
   End Dialog
   Dim QuitDialog As QuitDialogTemplate
   rc% = Dialog(QuitDialog)
End Sub
```

See Also CancelButton (statement); CheckBox (statement); ComboBox (statement); Dialog (function); Dialog (statement); DropListBox (statement); GroupBox (statement); ListBox (statement); OKButton (statement); OptionButton (statement); OptionGroup (statement); Picture (statement); PushButton (statement); Text (statement); TextBox (statement); DlgProc (function); HelpButton (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Boolean (data type)

Syntax `Boolean`

Description A data type capable of representing the logical values **True** and **False**.

Comments **Boolean** variables are used to hold a binary value—either **True** or **False**. Variables can be declared as **Boolean** using the **Dim**, **Public**, or **Private** statement.

Variants can hold **Boolean** values when assigned the results of comparisons or the constants **True** or **False**.

Internally, a **Boolean** variable is a 2-byte value holding –1 (for **True**) or 0 (for **False**).

Any type of data can be assigned to **Boolean** variables. When assigning, non-0 values are converted to **True**, and 0 values are converted to **False**. When converting strings to **Boolean**, BasicScript recognizes localized versions of the strings "True" and "False", converting these to the True and False respectively.

When appearing as a structure member, **Boolean** members require 2 bytes of storage.

When used within binary or random files, 2 bytes of storage are required.

When passed to external routines, **Boolean** values are sign-extended to the size of an integer on that platform (either 16 or 32 bits) before pushing onto the stack.

There is no type-declaration character for **Boolean** variables.

Boolean variables that have not yet been assigned are given an initial value of **False**.

See Also Currency (data type);Date (data type); Integer (data type); Long (data type); Object (data type); Single (data type); String (data type); Variant (data type); DefType (statement); CBool (function).

Platform(s) All.

ButtonEnabled (function)

Syntax `ButtonEnabled(`*name$* `|` *id*`)`

Description Returns **True** if the specified button within the current window is enabled; returns **False** otherwise.

Comments The **ButtonEnabled** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the push button.
id	**Integer** specifying the ID of the push button.

When a button is enabled, it can be clicked using the **SelectButton** statement.

Note: The **ButtonEnabled** function is used to determine whether a push button is enabled in another application's dialog box. Use the **DlgEnable** function to retrieve the enabled state of a push button in a dynamic dialog box.

Example

```
'This code fragment checks to see whether a button is enabled
'before clicking it.
Sub Main()
   If ButtonEnabled("Browse...") Then
      SelectButton "Browse..."
   Else
      MsgBox "Can't browse right now."
   End If
End Sub
```

See Also ButtonExists (function); SelectButton (statement).

Platform(s) Windows.

ButtonExists (function)

Syntax `ButtonExists(`*name$* `|` *id*`)`

Description Returns **True** if the specified button exists within the current window; returns **False** otherwise.

Comments The **ButtonExists** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the push button.
id	**Integer** specifying the ID of the push button.

Note: The **ButtonExists** function is used to determine whether a push button exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This code fragment selects the More button if it exists. If it
'does not exist, then this code fragment does nothing.
Sub Main()
   If ButtonExists("More >>") Then
      SelectButton "More >>"'Display more stuff.
   End If
End Sub
```

See Also ButtonEnabled (function); SelectButton (statement).

Platform(s) Windows.

ByRef (keyword)

Syntax `...,ByRef` *parameter*`,...`

Description Used within the **Sub**...**End Sub**, **Function**...**End Function**, or **Declare** statement to specify that a given parameter can be modified by the called routine.

Comments Passing a parameter by reference means that the caller can modify that variable's value.

Unlike the **ByVal** keyword, the **ByRef** keyword cannot be used when passing a parameter. The absence of the **ByVal** keyword is sufficient to force a parameter to be passed by reference:

```
MySub ByVal i     'Pass i by value.
MySub ByRef i     'Illegal (will not compile).
MySub i           'Pass i by reference.
```

Example

```
Sub Test(ByRef a As Variant)
```

```
   a = 14
End Sub
Sub Main()
   b = 12
   Test b
   MsgBox "The ByRef value is: " & b'Displays 14.
End Sub
```

See Also () (keyword); ByVal (keyword).

Platform(s) All.

ByVal (keyword)

Syntax `...ByVal` *parameter*`...`

Description Forces a parameter to be passed by value rather than by reference.

Comments The **ByVal** keyword can appear before any parameter passed to any function, statement, or method to force that parameter to be passed by value. Passing a parameter by value means that the caller cannot modify that variable's value.

Enclosing a variable within parentheses has the same effect as the **ByVal** keyword:

```
Foo ByVal i       'Forces i to be passed by value.
Foo(i)            'Forces i to be passed by value.
```

When calling external statements and functions (i.e., routines defined using the **Declare** statement), the **ByVal** keyword forces the parameter to be passed by value regardless of the declaration of that parameter in the **Declare** statement. The following example shows the effect of the **ByVal** keyword used to passed an **Integer** to an external routine:

```
Declare Sub Foo Lib "MyLib" (ByRef i As Integer)
i% = 6
Foo ByVal i%         'Pass a 2-byte Integer.
Foo i%               'Pass a 4-byte pointer to an Integer.
```

Since the **Foo** routine expects to receive a pointer to an **Integer**, the first call to **Foo** will have unpredictable results.

Example

```
'This example demonstrates the use of the ByVal keyword.
Sub Foo(a As Integer)
   a = a + 1
End Sub
Sub Main()
   Dim i As Integer
   i = 10
   Foo i
   MsgBox "The ByVal value is: " & i
      'Displays 11 (Foo changed the value).
```

```
   Foo ByVal i
   MsgBox "The ByVal value is still: " & i
      'Displays 11 (Foo did not change the value).
End Sub
```

See Also () (keyword); ByRef (keyword).

Platform(s) All.

Call (statement)

Syntax `Call` *subroutine_name* `[(`*arguments*`)]`

Description Transfers control to the given subroutine, optionally passing the specified arguments.

Comments Using this statement is equivalent to:

subroutine_name `[`*arguments*`]`

Use of the **Call** statement is optional. The **Call** statement can only be used to execute subroutines; functions cannot be executed with this statement. The subroutine to which control is transferred by the **Call** statement must be declared outside of the **Main** procedure, as shown in the following example.

Examples

```
'This example demonstrates the use of the Call statement to pass
'control to another function.
Sub Example_Call(s$)
   'This subroutine is declared externally to Main and displays
   'the text passed in the parameter s$.
   MsgBox "Call: " & s$
End Sub
Sub Main()
   'This example assigns a string variable to display, then
   'calls subroutine Example_Call, passing parameter S$ to be
   'displayed in a message box within the subroutine.
   s$ = "DAVE"
   Example_Call s$
   Call Example_Call("SUSAN")
End Sub
```

See Also **Goto** (statement); **GoSub** (statement); **Declare** (statement).

Platform(s) All.

CancelButton (statement)

Syntax `CancelButton` *x*, *y*, *width*, *height* `[,`*.Identifier*`]`

Description Defines a Cancel button that appears within a dialog box template.

Comments This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

Selecting the Cancel button (or pressing Esc) dismisses the user dialog box, causing the **Dialog** function to return 0. (Note: A dialog function can redefine this behavior.) Pressing the Esc key or double-clicking the close box will have no effect if a dialog box does not contain a **CancelButton** statement.

The **CancelButton** statement requires the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the dimensions of the control in dialog units.
.Identifier	Optional parameter specifying the name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). If this parameter is omitted, then the word "Cancel" is used.

A dialog box must contain at least one **OKButton**, **CancelButton**, or **PushButton** statement; otherwise, the dialog box cannot be dismissed.

Example

```
'This example creates a dialog box with OK and Cancel buttons.
Sub Main()
   Begin Dialog SampleDialogTemplate 37,32,48,52,"Sample"
      OKButton 4,12,40,14,.OK
      CancelButton 4,32,40,14,.Cancel
   End Dialog
   Dim SampleDialog As SampleDialogTemplate
   r% = Dialog(SampleDialog)
   If r% = 0 Then MsgBox "Cancel was pressed!"
End Sub
```

See Also **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

CBool (function)

Syntax `CBool(`*expression*`)`

Description Converts *expression* to **True** or **False**, returning a **Boolean** value.

Comments The *expression* parameter is any expression that can be converted to a **Boolean**. A runtime error is generated if *expression* is **Null**.

All numeric data types are convertible to **Boolean**. If *expression* is zero, then the **CBool** returns **False**; otherwise, **CBool** returns **True**. **Empty** is treated as **False**.

If *expression* is a **String**, then **CBool** first attempts to convert it to a number, then converts the number to a **Boolean**. A runtime error is generated if *expression* cannot be converted to a number.

A runtime error is generated if *expression* cannot be converted to a **Boolean**.

Example

```
'This example uses CBool to determine whether a string is
'numeric or just plain text.
Sub Main()
   Dim IsNumericOrDate As Boolean
   s$ = "34224.54"
   IsNumericOrDate = CBool(IsNumeric(s$) Or IsDate(s$))
   If IsNumericOrDate = True Then
      MsgBox s$ & " is either a valid date or number!"
   Else
      MsgBox s$ & " is not a valid date or number!"
   End If
End Sub
```

See Also **CCur** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Boolean** (data type).

Platform(s) All.

CCur (function)

Syntax `CCur(`*expression*`)`

Description Converts any expression to a **Currency**.

Comments This function accepts any expression convertible to a **Currency**, including strings. A runtime error is generated if *expression* is **Null** or a **String** not convertible to a number. **Empty** is treated as 0.

When passed a numeric expression, this function has the same effect as assigning the numeric expression number to a **Currency**.

When used with variants, this function guarantees that the variant will be assigned a **Currency** (**VarType** 6).

Example

```
'This example displays the value of a String converted into a
'Currency value.
Sub Main()
   i$ = "100.44"
   MsgBox "The currency value is: " & CCur(i$)
End Sub
```

See Also **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Currency** (data type).

Platform(s) All.

CDate, CVDate (functions)

Syntax
```
CDate(expression)
CVDate(expression)
```

Description Converts *expression* to a date, returning a **Date** value.

Comments The *expression* parameter is any expression that can be converted to a **Date**. A runtime error is generated if *expression* is **Null**.

If *expression* is a **String**, an attempt is made to convert it to a **Date** using the current country settings. If *expression* does not represent a valid date, then an attempt is made to convert *expression* to a number. A runtime error is generated if *expression* cannot be represented as a date.

These functions are sensitive to the date and time formats of your computer.

The **CDate** and **CVDate** functions are identical.

Example
```
'This example takes two dates and computes the difference
'between them.
Sub Main()
   Dim date1 As Date
   Dim date2 As Date
   Dim diff As Date
   date1 = CDate(#1/1/1994#)
   date2 = CDate("February 1, 1994")
   diff = DateDiff("d",date1,date2)
   MsgBox "The date difference is " & CInt(diff) & " days."
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Date** (data type).

Platform(s) All.

CDbl (function)

Syntax
```
CDbl(expression)
```

Description Converts any expression to a **Double**.

Comments This function accepts any expression convertible to a **Double**, including strings. A runtime error is generated if *expression* is **Null**. **Empty** is treated as 0.0.

When passed a numeric expression, this function has the same effect as assigning the numeric expression number to a **Double**.

When used with variants, this function guarantees that the variant will be assigned a **Double** (**VarType** 5).

Example

```
'This example displays the result of two numbers as a Double.
Sub Main()
   i% = 100
   j! = 123.44
   MsgBox "The double value is: " & CDbl(i% * j!)
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Double** (data type).

Platform(s) All.

ChDir (statement)

Syntax `ChDir` *path*

Description Changes the current directory of the specified drive to *path*.

Comments This routine will not change the current drive. (See **ChDrive** [statement].)

Example

```
'This example saves the current directory, then changes to the
'root directory, displays the old and new directories, restores
'the old directory, and displays it.
Const crlf = $(13) + Chr$(10)
Sub Main()
   save$ = CurDir$
   ChDir (Basic.PathSeparator$)
   MsgBox "Old: " & save$ & crlf & "New: " & CurDir$
   ChDir (save$)
   MsgBox "Directory restored to: " & CurDir$
End Sub
```

See Also **ChDrive** (statement); **CurDir**, **CurDir$** (functions); **Dir**, **Dir$** (functions); **MkDir** (statement); **RmDir** (statement); **DirList** (statement).

Platform(s) All.

Platform Notes **UNIX:** UNIX platforms do not support drive letters.

Platform Notes

NetWare: NetWare (and other operating systems) may not support the use of dots to indicate the current and parent directories unless configured to do so.

NetWare does not support drive letters. Directory specifications under NetWare use the following format:

volume:[*dir*\ [*dir*\]...]*file.ext*

The *volume* specification can be up to 14 characters.

Windows, Win32: BasicScript tracks and remembers the current directory for all drives in the system for that process.

Macintosh: The Macintosh does not support drive letters.

The Macintosh uses the colon (":") as the path separator. A double colon ("::") specifies the parent directory.

ChDrive (statement)

Syntax

`ChDrive` *drive*

Description

Changes the default drive to the specified drive.

Comments

Only the first character of *drive* is used.

Also, *drive* is not case-sensitive.

If *drive* is empty, then the current drive is not changed.

Example

```
'This example saves the current directory in CD, then extracts
'the current drive letter and saves it in Save$. If the current
'drive is D, then it is changed to C; otherwise, it is changed
'to D. Then the saved drive is restored and displayed.
Const crlf$ = Chr$(13) + Chr$(10)
Sub Main()
   cd$ = CurDir$
   save$ = Mid$(CurDir$,1,1)
   If save$ = "D" Then
      ChDrive("C")
   Else
      ChDrive("D")
   End If
   MsgBox "Old: " & save$ & crlf & "New: " & CurDir$
   ChDrive (save$)
   MsgBox "Directory restored to: " & CurDir$
End Sub
```

See Also

ChDir (statement); **CurDir**, **CurDir$** (functions); **Dir**, **Dir$** (functions); **MkDir** (statement); **RmDir** (statement); **DiskDrives** (statement).

Platform(s) Windows, Win32, NetWare. OS/2.

Platform Notes **UNIX, Macintosh:** UNIX platforms and the Macintosh do not support drive letters.

NetWare: Since NetWare does not support drive letters, the *drive* parameter specifies a volume name (up to 14 characters).

CheckBox (statement)

Syntax `CheckBox` *x*, *y*, *width*, *height*, *title$*, *.Identifier*

Description Defines a check box within a dialog box template.

Comments Check box controls are either on or off, depending on the value of *.Identifier*.

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **CheckBox** statement requires the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** coordinates specifying the dimensions of the control in dialog units.
title$	**String** containing the text that appears within the check box. This text may contain an ampersand character to denote an accelerator letter, such as "&Font" for Font (indicating that the Font control may be selected by pressing the F accelerator key).
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the state of the check box (1 = checked; 0 = unchecked). This variable can be accessed using the syntax: **DialogVariable**.*Identifier*.

When the dialog box is first created, the value referenced by *.Identifier* is used to set the initial state of the check box. When the dialog box is dismissed, the final state of the check box is placed into this variable. By default, the *.Identifier* variable contains 0, meaning that the check box is unchecked.

Example

```
'This example displays a dialog box with two check boxes in
'different states.
Sub Main()
   Begin Dialog SaveOptionsTemplate 36,32,151,52,"Save"
      GroupBox 4,4,84,40,"GroupBox"
```

```
      CheckBox 12,16,67,8,"Include heading",.IncludeHeading
      CheckBox 12,28,73,8,"Expand keywords",.ExpandKeywords
      OKButton 104,8,40,14,.OK
      CancelButton 104,28,40,14,.Cancel
   End Dialog
   Dim SaveOptions As SaveOptionsTemplate
   SaveOptions.IncludeHeading = 1'Check box initially on.
   SaveOptions.ExpandKeywords = 0'Check box initially off.
   r% = Dialog(SaveOptions)
   If r% = -1 Then
      MsgBox "OK was pressed."
   End If
End Sub
```

See Also **CancelButton** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); Begin **Dialog** (statement), **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, OS/2, Macintosh, UNIX.

Platform Notes **Windows, Win32, OS/2:** On Windows, Win32, and OS/2 platforms, accelerators are underlined, and the accelerator combination Alt+*letter* is used.

Macintosh: On the Macintosh, accelerators are normal in appearance, and the accelerator combination Command+*letter* is used..

CheckBoxEnabled (function)

Syntax `CheckBoxEnabled(`*name$* | *id*`)`

Description Returns **True** if the specified check box within the current window is enabled; returns **False** otherwise.

Comments The **CheckBoxEnabled** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the check box.
id	**Integer** specifying the ID of the check box.

When a check box is enabled, its state can be set using the **SetCheckBox** statement.

Note: The **CheckBoxEnabled** function is used to determine whether a check box is enabled in another application's dialog box. Use the **DlgEnable** function within dynamic dialog boxes.

Example

```
'This code checks to see whether a check box is enabled.
Sub Main()
   If CheckBoxEnabled("Portrait") Then
      SetCheckBox "Portrait",1
   End If
End Sub
```

See Also **CheckBoxExists** (function); **GetCheckBox** (function); **SetCheckBox** (statement).

Platform(s) Windows.

CheckBoxExists (function)

Syntax `CheckBoxExists(`*name$* `|` *id*`)`

Description Returns **True** if the specified check box exists within the current window; returns **False** otherwise.

Comments The **CheckBoxExists** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the check box.
id	**Integer** specifying the ID of the check box.

Note: The **CheckBoxExists** function is used to determine whether a check box exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This code fragment checks to ensure that the Portrait check
'box is selectable before selecting it.
Sub Main()
   If CheckBoxExists("Portrait") Then
      If CheckBoxEnabled("Portrait") Then
         SetCheckBox "Portrait",1
      End If
   End If
End Sub
```

See Also **CheckBoxEnabled** (function); **GetCheckBox** (function); **SetCheckBox** (statement).

Platform(s) Windows.

Choose (function)

Syntax `Choose(`*index*`,`*expression1*`,`*expression2*`,...,`*expression13*`)`

Description Returns the expression at the specified index position.

Comments The *index* parameter specifies which expression is to be returned. If *index* is 1, then *expression1* is returned; if *index* is 2, then *expression2* is returned, and so on. If *index* is less than 1 or greater than the number of supplied expressions, then **Null** is returned.

The *index* parameter is rounded down to the nearest whole number.

The **Choose** function returns the expression without converting its type. Each expression is evaluated before returning the selected one.

Example

```
'This example assigns a variable of indeterminate type to a.
Sub Main()
   Dim a As Variant
   Dim c As Integer
   c% = 2
   a = Choose(c%,"Hello, world",#1/1/94#,5.5,False)
   'Display the date passed as parameter 2.
   MsgBox "Item " & c% & " is '" & a & "'"
End Sub
```

See Also **Switch** (function); **IIf** (function); **If**...**Then**...**Else** (statement); **Select**...**Case** (statement).

Platform(s) All.

Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ (functions)

Syntax

```
Chr[$](charcode)
ChrB[$](charcode)
ChrW[$](charcode)
```

Description Returns the character whose value is *charcoode*.

Comments The **Chr$**, **ChrB$**, and **ChrW$** functions return a **String**, whereas the **Chr**, **ChrB**, and **ChrW** functions return a **String** variant.

These functions behave differently depending on the string format used by BasicScript. These differences are summarized in the following table:

Function	String Format	Description of charcode	Returns
Chr	SBCS	Value between 0 and 255	A 1-byte character string.
Chr$			
	MBCS	Value of an MBCS character between -32768 and 32767	A 1-byte or 2-byte MBCS character string depending on *charcode*.

Function	String Format	Description of charcode	Returns
	Wide	Value of an MBCS character between -32768 and 32767	A 2-byte character string.
ChrB	SBCS	Value between 0 and 255	A 1-byte character string.
ChrB$			
	MBCS	Value between 0 and 255	A 1-byte character string.
	Wide	Value between 0 and 255	A 1-byte character string.
ChrW	SBCS	Value between 0 and 255	A 1-byte character string (same as the **Chr** and **Chr$** functions)
ChrW$			
	MBCS	Value of an MBCS character between -32768 and 32767	A 1-byte or 2-byte MBCS character string depending on *charcode*.
	Wide	Value of a wide character between -32768 and 32767	A 2-byte character string.

The **Chr$** function can be used within constant declarations, as in the following example:

```
Const crlf = Chr$(13) + Chr$(10)
```

Some common uses of this function are:

Chr$(9)	Tab
Chr$(13) + Chr$(10)	End-of-line (carriage return, linefeed)
Chr$(26)	End-of-file
Chr$(0)	Null

Examples

```
Sub Main()
   'Concatenates carriage return (13) and line feed (10) to
   'CRLF$, then displays a multiple-line message using CRLF$ to
   'separate lines.
   crlf$ = Chr$(13) + Chr$(10)
   MsgBox "First line." & crlf$ & "Second line."
   'Fills an array with the ASCII characters for ABC and
   'displays their corresponding characters.
   Dim a%(2)
   For i = 0 To 2
      a%(i) = (65 + i)
   Next i
```

```
   MsgBox "The first three elements of the array are: " _
      & Chr$(a%(0)) & Chr$(a%(1)) & Chr$(a%(2))
End Sub
```

See Also **Asc** (function); **Str**, **Str$** (functions).

Platform(s) All.

CInt (function)

Syntax `CInt(`*expression*`)`

Description Converts *expression* to an **Integer**.

Comments This function accepts any expression convertible to an **Integer**, including strings. A runtime error is generated if *expression* is **Null**. **Empty** is treated as 0.

The passed numeric expression must be within the valid range for integers:

```
-32768 <= expression <= 32767
```

A runtime error results if the passed expression is not within the above range.

When passed a numeric expression, this function has the same effect as assigning a numeric expression to an **Integer**. Note that integer variables are rounded before conversion.

When used with variants, this function guarantees that the expression is converted to an **Integer** variant (**VarType** 2).

Example

```
'This example demonstrates the various results of integer
'manipulation with CInt.
Sub Main()
   '(1) Assigns i# to 100.55 and displays its integer
   'representation (101).
   i# = 100.55
   MsgBox "The value of CInt(i) = " & CInt(i#)
   '(2) Sets j# to 100.22 and displays the CInt representation
   '(100).
   j# = 100.22
   MsgBox "The value of CInt(j) = " & CInt(j#)
   '(3) Assigns k% (integer) to the CInt sum of j# and k% and
   'displays k% (201).
   k% = CInt(i# + j#)
   MsgBox "The integer sum of 100.55 and 100.22 is: " & k%
   '(4) Reassigns i# to 50.35 and recalculates k%, then
   'displays the result (note rounding).
   i# = 50.35
   k% = CInt(i# + j#)
   MsgBox "The integer sum of 50.35 and 100.22 is: " & k%
```

```
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Integer** (data type).

Platform(s) All.

Clipboard$ (function)

Syntax `Clipboard$[()]`

Description Returns a **String** containing the contents of the Clipboard.

Comments If the Clipboard doesn't contain text or the Clipboard is empty, then a zero-length string is returned.

Example

```
'This example puts text on the Clipboard, displays it, clears
'the Clipboard, and displays the Clipboard again.
Const crlf = Chr$(13) + Chr$(10)

Sub Main()
   Clipboard$ "Hello out there!"
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
   Clipboard.Clear
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

See Also **Clipboard$** (statement); **Clipboard.GetText** (method); **Clipboard.SetText** (method).

Platform(s) Windows, Win32, Macintosh, OS/2.

Clipboard$ (statement)

Syntax `Clipboard$` *NewContent$*

Description Copies *NewContent$* into the Clipboard.

Example

```
'This example puts text on the Clipboard, displays it, clears
'the Clipboard, and displays the Clipboard again.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Clipboard$ "Hello out there!"
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
   Clipboard.Clear
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

See Also **Clipboard$** (function); **Clipboard.GetText** (method); **Clipboard.SetText** (method).

Platform(s) Windows, Win32, Macintosh, OS/2.

Clipboard.Clear (method)

Syntax
```
Clipboard.Clear
```

Description This method clears the Clipboard by removing any content.

Example
```
'This example puts text on the Clipboard, displays it, clears
'the Clipboard, and displays the Clipboard again.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Clipboard$ "Hello out there!"
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
   Clipboard.Clear
   MsgBox "The text in the Clipboard is:" & crlf & Clipboard$
End Sub
```

Platform(s) Windows, Win32, Macintosh, OS/2.

Clipboard.GetFormat (method)

Syntax *WhichFormat* = `Clipboard.GetFormat(`*format*`)`

Description Returns **True** if data of the specified format is available in the Clipboard; returns **False** otherwise.

Comments This method is used to determine whether the data in the Clipboard is of a particular format. The format parameter is an **Integer** representing the format to be queried:

Format	Value	Description
ebCFText	1	Text
ebCFBitmap	2	Bitmap
ebCFMetafile	3	Metafile
ebCFDIB	8	Device-independent bitmap (DIB)
ebCFPalette	9	Color palette
ebCFUnicodeText	13	Unicode text

Example
```
'This example puts text on the Clipboard, checks whether there
'is text on the Clipboard, and if there is, displays it.
Sub Main()
   Clipboard$ "Hello out there!"
```

```
   If Clipboard.GetFormat(ebCFText) Then
      MsgBox Clipboard$
   Else
      MsgBox "There is no text in the Clipboard."
   End If
End Sub
```

See Also **Clipboard$** (function); **Clipboard$** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2.

Clipboard.GetText (method)

Syntax *text$* `= Clipboard.GetText([`*format*`])`

Description Returns the text contained in the Clipboard.

Comments The format parameter, if specified, must be **ebCFText** (1).

Example

```
'This example retrieves the text from the Clipboard and checks
'to make sure that it contains the word "dog."
Option Compare Text
Sub Main()
   If Clipboard.GetFormat(1) Then
      If Instr(Clipboard.GetText(1),"dog",1) = 0 Then
         MsgBox "The Clipboard doesn't contain the word ""dog."""
      Else
         MsgBox "The Clipboard contains the word ""dog""."
      End If
   Else
      MsgBox "The Clipboard does not contain text."
   End If
End Sub
```

See Also **Clipboard$** (statement); **Clipboard$** (function); **Clipboard.SetText** (method).

Platform(s) Windows, Win32, Macintosh, OS/2.

Platform Notes **Win32:** Under Win32, the *format* parameter must be either **ebCFText** or **ebCFUnicodeText**. If the *format* parameter is omitted, then BasicScript first looks for text of the specified type depending on the platform:

Platform	Clipboard Format
Windows NT	UNICODE
Windows 95	MBCS
Win32s	MBCS

Clipboard.SetText (method)

Syntax `Clipboard.SetText` *data$* `[,`*format*`]`

Description Copies the specified text string to the Clipboard.

Comments The *data$* parameter specifies the text to be copied to the Clipboard. The *format* parameter, if specified, must be **ebCFText** (1).

Example

```
'This example gets the contents of the Clipboard and uppercases
'it.
Sub Main()
   If Not Clipboard.GetFormat(1) Then Exit Sub
   Clipboard.SetText UCase$(Clipboard.GetText(1)),1
End Sub
```

See Also **Clipboard$** (statement); **Clipboard.GetTex**t (method); **Clipboard$** (function).

Platform(s) Windows, Win32, Macintosh, OS/2.

Platform Notes **Win32:** Under Win32, the *format* parameter must be either **ebCFText** or **ebCFUnicodeText**. If the *format* parameter is omitted, then BasicScript places the text into the clipboard in the following format depending on the platform.

Platform	Clipboard Format
Windows NT	UNICODE
Windows 95	MBCS
Win32s	MBCS

CLng (function)

Syntax `CLng(`*expression*`)`

Description Converts *expression* to a **Long**.

Comments This function accepts any expression convertible to a **Long**, including strings. A runtime error is generated if *expression* is **Null**. **Empty** is treated as 0.

The passed expression must be within the following range:

```
-2147483648 <= expression <= 2147483647
```

A runtime error results if the passed expression is not within the above range.

When passed a numeric expression, this function has the same effect as assigning the numeric expression to a **Long**. Note that long variables are rounded before conversion.

When used with variants, this function guarantees that the expression is converted to a Long variant (**VarType** 3).

Example

```
'This example displays the results for various conversions of i
'and j (note rounding).
Sub Main()
      i% = 100
      j& = 123.666
      MsgBox "The result is: " & CLng(i% * j&)'Displays 12367.
      MsgBox "The variant type is: " & Vartype(CLng(i%))
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CSng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Long** (data type).

Platform(s) All.

Close (statement)

Syntax `Close [[#] filenumber [,[#] filenumber]...]`

Description Closes the specified files.

Comments If no arguments are specified, then all files are closed.

Example

```
'This example opens four files and closes them in various
'combinations.
Sub Main()
   Open "test1" For Output As #1
   Open "test2" For Output As #2
   Open "test3" For Random As #3
   Open "test4" For Binary As #4
   MsgBox "The next available file number is :" & FreeFile()
   Close #1    'Closes file 1 only.
   Close #2, #3'Closes files 2 and 3.
   Close          'Closes all remaining files(4).
   MsgBox "The next available file number is :" & FreeFile()
End Sub
```

See Also **Open** (statement); **Reset** (statement); **End** (statement).

Platform(s) All.

ComboBox (statement)

Syntax `ComboBox x,y,width,height,ArrayVariable,.Identifier`

Description This statement defines a combo box within a dialog box template.

Comments

When the dialog box is invoked, the combo box will be filled with the elements from the specified array variable.

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **ComboBox** statement requires the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the dimensions of the control in dialog units.
ArrayVariable	Single-dimensioned array used to initialize the elements of the combo box. If this array has no dimensions, then the combo box will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. *ArrayVariable* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates a string variable whose value corresponds to the content of the edit field of the combo box. This variable can be accessed using the syntax: *DialogVariable.Identifier*.

When the dialog box is invoked, the elements from *ArrayVariable* are placed into the combo box. The *.Identifier* variable defines the initial content of the edit field of the combo box. When the dialog box is dismissed, the *.Identifier* variable is updated to contain the current value of the edit field.

Example

```
'This example creates a dialog box that allows the user to
'select a day of the week.
Sub Main()
   Dim days$(6)
   days$(0) = "Monday"
   days$(1) = "Tuesday"
   days$(2) = "Wednesday"
   days$(3) = "Thursday"
   days$(4) = "Friday"
   days$(5) = "Saturday"
   days$(6) = "Sunday"
   Begin Dialog DaysDialogTemplate 16,32,124,96,"Days"
```

```
      OKButton 76,8,40,14,.OK
      Text 8,10,39,8,"&Weekdays:"
      ComboBox 8,20,60,72,days$,.Days
   End Dialog
   Dim DaysDialog As DaysDialogTemplate
   DaysDialog.Days = "Tuesday"
   r% = Dialog(DaysDialog)
   MsgBox "You selected: " & DaysDialog.Days
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement), **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

ComboBoxEnabled (function)

Syntax `ComboBoxEnabled(`*name$* `|` *id*`)`

Description Returns **True** if the specified combo box is enabled within the current window or dialog box; returns **False** otherwise.

Comments The **ComboBoxEnabled** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the combo box. The name of a combo box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a combo box. A runtime error is generated if a combo box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the combo box.

A runtime error is generated if the specified combo box does not exist.

Note: The **ComboBoxEnabled** function is used to determine whether a combo box is enabled in another application's dialog box. Use the **DlgEnable** function in dynamic dialog boxes.

Example

```
'This example checks to see whether a combo box is active. If it
'is, then it inserts some text into it.
Sub Main()
      If ComboBoxEnabled("Filename:") Then
```

```
         SelectComboBoxItem "Filename:","sample.txt"
      End If
      If ComboBoxEnabled(365) Then
         SelectComboBoxItem 365,3'Select the third item.
      End If
End Sub
```

See Also **ComboBoxExists** (function); **GetComboBoxItem**$ (function); **GetComboBoxItemCount** (function); **SelectComboBoxItem** (statement).

Platform(s) Windows.

ComboBoxExists (function)

Syntax `ComboBoxExists(`*name$* `|` *id*`)`

Description Returns **True** if the specified combo box exists within the current window or dialog box; returns **False** otherwise.

Comments The **ComboBoxExists** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the combo box. The name of a combo box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a combo box. A runtime error is generated if a combo box with that name cannot be found within the active window
id	**Integer** specifying the ID of the combo box.

Note: The **ComboBoxExists** function is used to determine whether a combo box exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This code fragment checks to ensure that a combo box exists and
'is enabled before selecting the last item.
Sub Main()
      If ComboBoxExists("Filename:") Then
         If ComboBoxEnabled("Filename:") Then
            NumItems = GetComboBoxItemCount("Filename:")
            SelectComboBoxItem "Filename:",NumItems
         End If
      End If
End Sub
```

See Also **ComboBoxEnabled** (function); **GetComboBoxItem**$ (function); **GetComboBoxItemCount** (function); **SelectComboBoxItem** (statement).

Platform(s) Windows.

Command, Command$ (functions)

Syntax
```
Command[$][()]
```

Description Returns the argument from the command line used to start the application.

Comments **Command$** returns a string, whereas **Command** returns a **String** variant.

Example
```
'This example gets the command line and parameters, checks to
'see whether the string "/s" is present, and displays the result.
Sub Main()
      cmd$ = Command$
      If (InStr(cmd$,"/s")) <> 0 Then
         MsgBox "Application was started with the /s switch."
      Else
         MsgBox "Application was started without the /s switch."
      End If
      If cmd$ <> "" Then
         MsgBox "The command line startup options were: " & cmd$
      Else
         MsgBox "No command line startup options were used!"
      End If
End Sub
```

See Also **Environ**, **Environ$** (functions).

Platform(s) All.

Comments (topic)

Comments can be added to BasicScript code in the following manner:

All text between a single quotation mark and the end of the line is ignored:
```
MsgBox "Hello"       'Displays a message box.
```

The **REM** statement causes the compiler to ignore the entire line:
```
REM This is a comment.
```

BasicScript supports C-style multiline comment blocks /*...*/, as shown in the following example:
```
MsgBox "Before comment"
/* This stuff is all commented out.
This line, too, will be ignored.
```

```
This is the last line of the comment. */
MsgBox "After comment"
```

Note: C-style comments can be nested.

Comparison Operators (topic)

Syntax *expression1* [< | > | <= | >= | <> | =] *expression2*

Description Comparison operators return **True** or **False** depending on the operator.

Comments The comparison operators are listed in the following table:

Operator	Returns True If
>	*expression1* is greater than *expression2*
<	*expression1* is less than *expression2*
<=	*expression1* is less than or equal to *expression2*
>=	*expression1* is greater than or equal to *expression2*
<>	*expression1* is not equal to *expression2*
=	*expression1* is equal to *expression2*

This operator behaves differently depending on the types of the expressions, as shown in the following table:

If one expression is	And the other expression is	Then
Numeric	Numeric	A numeric comparison is performed (see below).
String	**String**	A string comparison is performed (see below).
Numeric	**String**	A compile error is generated.
Variant	**String**	A string comparison is performed (see below).
Variant	Numeric	A variant comparison is performed (see below).
Null variant	Any data type	Returns **Null**.
Variant	**Variant**	A variant comparison is performed (see below).

String Comparisons

If the two expressions are strings, then the operator performs a text comparison between the two string expressions, returning **True** if *expression1* is less than *expression2*. The text comparison is case-sensitive if **Option Compare** is **Binary**; otherwise, the comparison is case-insensitive.

When comparing letters with regard to case, lowercase characters in a string sort greater than uppercase characters, so a comparison of "a" and "A" would indicate that "a" is greater than "A".

Numeric Comparisons

When comparing two numeric expressions, the less precise expression is converted to be the same type as the more precise expression.

Dates are compared as doubles. This may produce unexpected results as it is possible to have two dates that, when viewed as text, display as the same date when, in fact, they are different. This can be seen in the following example:

```
Sub Main()
   Dim date1 As Date
   Dim date2 As Date
   date1 = Now
   date2 = date1 + 0.000001      'Adds a fraction of a
                                 'second.
   MsgBox date2 = date1          'Prints False (the dates are
                                 'different).
   MsgBox date1 & "," & date2    'Prints two dates that are
                                 'the same.
End Sub
```

Variant Comparisons

When comparing variants, the actual operation performed is determined at execution time according to the following table:

If one variant is	And the other variant is	Then
Numeric	Numeric	Compares the variants as numbers.
String	**String**	Compares the variants as text.
Numeric	**String**	The number is less than the string.
Null	Any other data type	**Null**.
Numeric	**Empty**	Compares the number with 0.
String	**Empty**	Compares the string with a zero-length string.

Examples

```
Sub Main()
   'Tests two literals and displays the result.
   If 5 < 2 Then
      MsgBox "5 is less than 2."
   Else
      MsgBox "5 is not less than 2."
   End If
   'Tests two strings and displays the result.
   If "This" < "That" Then
      MsgBox "'This' is less than 'That'."
   Else
      MsgBox "'That' is less than 'This'."
   End If
End Sub
```

See Also Operator Precedence (topic); **Is** (operator); **Like** (operator); **Option Compare** (statement).

Platform(s) All.

Const (statement)

Syntax `Const` *name* `[As` *type*`]` `=` *expression* `[,`*name* `[As` *type*`]` `=` *expression*`]...`

Description Declares a constant for use within the current script.

Comments The *name* is only valid within the current BasicScript script. Constant names must follow these rules:

1. Must begin with a letter.
2. May contain only letters, digits, and the underscore character.
3. Must not exceed 80 characters in length.
4. Cannot be a reserved word.

Constant names are not case-sensitive.

The *expression* must be assembled from literals or other constants. Calls to functions are not allowed except calls to the **Chr$** function, as shown below:

```
Const s$ = "Hello, there" + Chr(44)
```

Constants can be given an explicit type by declaring the *name* with a type-declaration character, as shown below:

```
Const a% = 5      'Constant Integer whose value is 5
Const b# = 5      'Constant Double whose value is 5.0
Const c$ = "5"    'Constant String whose value is "5"
Const d! = 5      'Constant Single whose value is 5.0
```

```
Const e& = 5      'Constant Long whose value is 5
```

The type can also be given by specifying the **As** *type* clause:

```
Const a As Integer = 5  'Constant Integer whose value is 5
Const b As Double = 5   'Constant Double whose value is 5.0
Const c As String = "5" 'Constant String whose value is "5"
Const d As Single = 5   'Constant Single whose value is 5.0
Const e As Long = 5     'Constant Long whose value is 5
```

You cannot specify both a type-declaration character and the *type*:

```
Const a% As Integer = 5 'THIS IS ILLEGAL.
```

If an explicit type is not given, then BasicScript will choose the most imprecise type that completely represents the data, as shown below:

```
Const a = 5             'Integer constant
Const b = 5.5           'Single constant
Const c = 5.5E200       'Double constant
```

Constants defined within a **Sub** or **Function** are local to that subroutine or function. Constants defined outside of all subroutines and functions can be used anywhere within that script. The following example demonstrates the scoping of constants:

```
Const DefFile = "default.txt"
Sub Test1
   Const DefFile = "foobar.txt"
   MsgBox DefFile    'Displays "foobar.txt".
End Sub
Sub Test2
   MsgBox DefFile    'Displays "default.txt".
End Sub
```

Example

```
'This example displays the declared constants in a dialog box
'(crlf produces a new line in the dialog box).
Const crlf = Chr$(13) + Chr$(10)
Const s As String = "This is a constant."
Sub Main()
      MsgBox s$ & crlf & "The constants are shown above."
End Sub
```

See Also **Def***Type* (statement); **Let** (statement); **=** (statement); Constants (topic).

Platform(s) All.

Constants (topic)

Constants are variables that cannot change value during script execution. The following constants are predefined by BasicScript.

Application State Constants (Used with AppSetState and AppGetState)

Constant	Value	Description
ebMinimized	1	The application is minimized.
ebMaximized	2	The application is maximized.
ebRestored	3	The application is restored.

BasicScript Constants

Constant	Value	Description
True	-1	Boolean value **True**.
False	0	Boolean value **False**.
Empty	**Empty**	Variant of type 0, indicating that the variant is uninitialized.
Nothing	0	Value indicating that an object variable no longer references a valid object.
Null	**Null**	Variant of type 1, indicating that the variant contains no data.

Character Constants

Constant	Value	Description
ebBack	**Chr$(8)**	String containing a backspace.
ebCr	**Chr$(13)**	String containing a carriage return.
ebCrLf	**Chr$(13) & Chr$(10)**	String containing a carriage-return linefeed pair.
ebFormFeed	**Chr$(11)**	String containing a form feed.
ebLf	**Chr$(10)**	String containing a line feed.
ebNullChar	**Chr$(0)**	String containing a single null character.

Character Constants

Constant	Value	Description
ebNullString	**0**	Special string value used to pass null pointers to external routines.
ebTab	**Chr$(9)**	String containing a tab.
ebVerticalTab	**Chr$(12)**	String containing a vertical tab.

Clipboard Constants (Used with Clipboard.GetText, Clipboard.SetText, and Clipboard.GetFormat)

Constant	Value	Description
ebCFText	1	Text.
ebCFBitmap	2	Bitmap.
ebCFMetafile	3	Metafile.
ebCFDIB	8	Device-independent bitmap.
ebCFPalette	9	Palette.
ebCFUnicode	13	Unicode text.

Compiler Constants

Constant	Value
AIX	True if development environment is AIX.
HPUX	True if development environment is HPUX.
Irix	True if development environment is Irix.
LINUX	True if development environment is LINUX.
Macintosh	True if development environment is Macintosh (680x0 or PowerPC).
MacPPC	True if development environment is PowerMac.
Mac68K	True if development environment is 68K Macintosh.
Netware	True if development environment is NetWare.
OS2	True if development environment is OS/2.

Compiler Constants (Continued)

Constant	Value
OSF1	True if development environment is OSF/1.
SCO	True if development environment is SCO.
Solaris	True if development environment is Solaris.
SunOS	True if development environment is SunOS.
UNIX	True if development environment is any UNIX platform.
UnixWare	True if development environment is UnixWare.
VMS	True if development environment is VMS.
Win16	True if development environment is 16-bit Windows.
Win32	True if development environment is 32-bit Windows.
Empty	**Empty**
False	**False**
Null	**Null**
True	**True**

Date Constants (Used with WeekDay, Format, DateAdd, DateDiff)

Constant	Value	Description
ebUseSunday	0	Use the date setting as specified by the current locale.
ebSunday	1	Sunday.
ebMonday	2	Monday.
ebTuesday	3	Tuesday.
ebWednesday	4	Wednesday.
ebThursday	5	Thursday.
ebFriday	6	Friday.
ebSaturday	7	Saturday.
ebFirstJan1	1	Start with week in which January 1 occurs.

Date Constants (Used with WeekDay, Format, DateAdd, DateDiff) (Continued)

Constant	Value	Description
ebFirstFourDays	2	Start with first week with at least four days in the new year.
ebFirstFullWeek	3	Start with first full week of the year.

File Constants (Used with Dir, Dir$, FileList, SetAttr, GetAttr, FileAttr)

Constant	Value	Description
ebNormal	0	Read-only, archive, subdir, and none.
ebReadOnly	1	Read-only files.
ebHidden	2	Hidden files.
ebSystem	4	System files.
ebVolume	8	Volume labels.
ebDirectory	16	Subdirectory.
ebArchive	32	Files that have changed since the last backup.
ebNone	64	Files with no attributes.

File Type Constants (Used with AppType and FileType)

Constant	Value	Description
ebDOS	1	A DOS executable file.
ebWindows	2	A Windows executable file.

Font Constants (Used with Text and TextBox)

Constant	Value	Description
ebRegular	1	Normal font (i.e., neither bold nor italic).
ebItalic	2	Italic font.
ebBold	4	Bold font.
ebBoldItalic	6	Bold-italic font.

IMEStat Constants (Returned by the IMEStat Function)

Constant	Value	Description
ebIMENoOp	0	IME not installed.
ebIMEOn	1	IME on.
ebIMEOff	2	IME off.
ebIMEDisabled	3	IME disabled.
ebIMEHiragana	4	Hiragana double-byte character.
ebIMEKatakanaDbl	5	Katakana double-byte characters.
ebIMEKatakanaSng	6	Katakana single-byte characters.
ebIMEAlphaDbl	7	Alphanumeric double-byte characters.
ebIMEAlphaSng	8	Alphanumeric single-byte characters.

Math Constants

Constant	Value	Description
PI	3.1415...	Value of PI.

MsgBox Constants

Constant	Value	Description
ebOKOnly	0	Displays only the OK button.
ebOKCancel	1	Displays OK and Cancel buttons.
ebAbortRetryIgnore	2	Displays Abort, Retry, and Ignore buttons.
ebYesNoCancel	3	Displays Yes, No, and Cancel buttons.
ebYesNo	4	Displays Yes and No buttons.
ebRetryCancel	5	Displays Cancel and Retry buttons.
ebCritical	16	Displays the stop icon.
ebQuestion	32	Displays the question icon.
ebExclamation	48	Displays the exclamation icon.
ebInformation	64	Displays the information icon.

MsgBox Constants (Continued)

Constant	Value	Description
ebApplicationModal	0	The current application is suspended until the dialog box is closed.
ebDefaultButton1	0	First button is the default button.
ebDefaultButton2	256	Second button is the default button.
ebDefaultButton3	512	Third button is the default button.
ebSystemModal	4096	All applications are suspended until the dialog box is closed.
ebOK	1	Returned from **MsgBox** indicating that OK was pressed.
ebCancel	2	Returned from **MsgBox** indicating that Cancel was pressed.
ebAbort	3	Returned from **MsgBox** indicating that Abort was pressed.
ebRetry	4	Returned from **MsgBox** indicating that Retry was pressed.
ebIgnore	5	Returned from **MsgBox** indicating that Ignore was pressed.
ebYes	6	Returned from **MsgBox** indicating that Yes was pressed.
ebNo	7	Returned from **MsgBox** indicating that No was pressed.

Platform Constants (Returned by Basic.OS)

Constant	Value	Description
ebWin16	0	Microsoft Windows (16-bit).
ebWin32	2	Microsoft Windows 95
		Microsoft Windows NT Workstation
		Microsoft Windows NT Server
		Microsoft Win32s running under Windows 3.1

Platform Constants (Returned by Basic.OS) (Continued)

Constant	Value	Description
ebSolaris	3	Sun Solaris 2.x
ebSunOS	4	SunOS
ebHPUX	5	HP-UX
ebIrix	7	Silicon Graphics IRIX
ebAIX	8	IBM AIX
ebNetware	9	Novell Netware
ebMacintosh	10	Apple Macintosh
ebOS2	11	IBM OS/2
ebSCO	13	SCO UNIX
ebUnixWare	14	Novell UnixWare
ebOSF1	15	OSF/1
ebVMS	16	VMS
ebLINUX	17	LINUX

Printer Constants (Used with PrinterSetOrientation and PrinterGetOrientation)

Constant	Value	Description
ebLandscape	1	Landscape paper orientation.
ebPortrait	2	Portrait paper orientation.

Que Constants (Used with the Que Statements)

Constant	Value	Description
ebLeftButton	1	Left mouse button.
ebRightButton	2	Right mouse button.

Shell Constants (Used with the Shell Function)

Constant	Value	Description
ebHide	0	Application is initially hidden.

Shell Constants (Used with the Shell Function) (Continued)

Constant	Value	Description
ebNormalFocus	1	Application is displayed at the default position and has the focus.
ebMinimizedFocus	2	Application is initially minimized and has the focus.
ebMaximizedFocus	3	Application is maximized and has the focus.
ebNormalNoFocus	4	Application is displayed at the default position and does not have the focus.
ebMinimizedNoFocus	6	Application is minimized and does not have the focus.

String Conversion Constants (Used with the StrConv Function)

Constant	Value	Description
ebUpperCase	1	Converts string to uppercase.
ebLowerCase	2	Converts string to lowercase.
ebProperCase	3	Capitalizes the first letter of each word.
ebWide	4	Converts narrow characters to wide characters.
ebNarrow	8	Converts wide characters to narrow characters.
ebKatakana	16	Converts Hiragana characters to Katakana characters.
ebHiragana	32	Converts Katakana characters to Hiragana characters.
ebUnicode	64	Converts string from MBCS to UNICODE.
ebFromUnicode	128	Converts string from UNICODE to MBCS.

Variant Constants (Returned by the VarType Function)

Constant	Value	Description
ebEmpty	0	Variant has not been initialized.
ebNull	1	Variant contains no valid data.

Variant Constants (Returned by the VarType Function) (Continued)

Constant	Value	Description
ebInteger	2	Variant contains an **Integer**.
ebLong	3	Variant contains a **Long**.
ebSingle	4	Variant contains a **Single**.
ebDouble	5	Variant contains a **Double**.
ebCurrency	6	Variant contains a **Currency**.
ebDate	7	Variant contains a **Date**.
ebString	8	Variant contains a **String**.
ebObject	9	Variant contains an **Object**.
ebError	10	Variant contains an **Error**.
ebBoolean	11	Variant contains a **Boolean**.
ebVariant	12	Variant contains an array of **Variants**.
ebDataObject	13	Variant contains a data object.
ebArray	8192	Added to any of the other types to indicate an array of that type.

You can define your own constants using the **Const** statement.

Preprocessor constants are defined using **#Const**.

Cos (function)

Syntax `Cos(`*number*`)`

Description Returns a **Double** representing the cosine of *number*.

Comments The *number* parameter is a **Double** specifying an angle in radians.

Example

```
'This example assigns the cosine of pi/4 radians (45 degrees)
'to C# and displays its value.
Sub Main()
      c# = Cos(3.14159 / 4)
      MsgBox "The cosine of 45 degrees is: " & c#
End Sub
```

See Also **Tan** (function); **Sin** (function); **Atn** (function).

Platform(s) All.

CreateObject (function)

Syntax `CreateObject(`*class*`)`

Description Creates an OLE Automation object and returns a reference to that object.

Comments The *class* parameter specifies the application used to create the object and the type of object being created. It uses the following syntax:

`"`*application.class*`",`

where *application* is the application used to create the object and *class* is the type of the object to create.

At runtime, **CreateObject** looks for the given application and runs that application if found. Once the object is created, its properties and methods can be accessed using the dot syntax (e.g., *object.property = value*).

There may be a slight delay when an automation server is loaded (this depends on the speed with which a server can be loaded from disk). This delay is reduced if an instance of the automation server is already loaded.

Examples

```
'This first example instantiates Microsoft Excel. It then uses
'the resulting object to make Excel visible and then close Excel.
Sub Main()
   Dim Excel As Object
   On Error GoTo Trap1     'Set error trap.
   Set Excel = CreateObject("excel.application")
   Excel.Visible = True 'Make Excel visible.
   Sleep 5000                    'Wait 5 seconds.
   Excel.Quit                    'Close Excel.
   Exit Sub                         'Exit before error trap.
Trap1:
   MsgBox "Can't create Excel object."'Display error message.
   Exit Sub                         'Reset error handler.
End Sub
'This second example uses CreateObject to instantiate a Visio
'object. It then uses the resulting object to create a new
'document.
Sub Main()
      Dim Visio As Object
      Dim doc As Object
      Dim page As Object
      Dim shape As Object
      Set Visio = CreateObject("visio.application")
```

```
    'Create Visio object.
    Set doc = Visio.Documents.Add("")'Create a new document.
    Set page = doc.Pages(1)          'Get first page.
    Set shape = page.DrawRectangle(1,1,4,4)
    shape.text = "Hello, world."     'Set text within shape.
End Sub
```

See Also **GetObject** (function); **Object** (data type).

Platform(s) Windows, Win32, Macintosh.

Cross-Platform Scripting (topic)

This section discusses different techniques that can be used to ensure that a given script runs on all platforms that support BasicScript.

Querying the Platform

A script can query the platform in order to take appropriate actions for that platform. This is done using the **Basic.OS** property. The following example uses this method to display a message to the user:

```
Sub Main()
   If Basic.OS = ebWindows Then
      MsgBox "This is a message."
   Else
      Print "This is a message."
   End If
End Sub
```

Querying the Capabilities of a Platform

Some capabilities of the current platform can be determined using the **Basic.Capability** method. This method takes a number indicating which capability is being queried and returns either **True** or **False** depending on whether that capability is or is not supported on the current platform. The following example uses this technique to read hidden files:

```
Sub Main()
   If Basic.Capability(3) Then
      f$ = Dir$("*",ebHidden)    'Hidden files supported.
   Else
      f$ = Dir$("*")             'Hidden files not
                                 'supported.
   End If
   'Print all the files.
   While f$ <> ""
```

```
      x = x + 1
      MsgBox "Matching file " & x & " is: " & f$
      f$ = Dir$
   Wend
End Sub
```

Byte Ordering with Files

One of the main differences between platforms is byte ordering. On some platforms, the processor requires that the bytes that make up a given data item be reversed from their expected ordering.

Byte ordering becomes problematic if binary data is transferred from one platform to another. This can only occur when writing data to files. For this reason, it is strongly recommended that files that are to be transported to a different platform with different byte ordering be sequential (i.e., do not use **Binary** and **Random** files).

If a **Binary** or **Random** file needs to be transported to another platform, you will have to take into consideration the following:

1. You must either decide on a byte ordering for your file or write information to the file indicating its byte ordering so that it can be queried by the script that is to read the file.
2. When reading a file on a platform in which the byte ordering matches, nothing further needs to be done. If the byte ordering is different, then the bytes of each data item read from a file need to be reversed. This is a difficult proposition.

Byte Ordering with Structures

Due to byte ordering differences between platforms, structure copying using the **LSet** statement produces different results. Consider the following example:

```
Type TwoInts
   first As Integer
   second As Integer
End Type
Type OneLong
   first As Long
End Type
Sub Main()
   Dim l As OneLong
   Dim i As TwoInts
   l.First = 4
   LSet i = l
   MsgBox "First integer: " & i.first
   MsgBox "Second integer: " & i.second
```

```
End Sub
```

On Intel-based platforms, bytes are stored in memory with the most significant byte first (known as little-endian format). Thus, the above example displays two dialog boxes, the first one displaying the number 4 and the second displaying the number 0.

On UNIX and Macintosh platforms, bytes are stored in memory with the least significant byte first (known as big-endian format). Thus, the above example displays two dialog boxes, the first one displaying the number 0 and the second displaying the number 4.

Scripts that rely on binary images of data must take the byte ordering of the current platform into account.

Reading and Writing to Text Files

Different platforms use different characters to represent end-of-line in a file. For example, under Windows, a carriage-return/linefeed pair is used. Under UNIX, a line feed by itself is used. On the Macintosh, a carriage return is used.

BasicScript takes this into account when reading text files. The following combinations are recognized and interpreted as end-of-line:

Carriage return	**Chr(13)**
Carriage return/line feed	**Chr(13) + Chr(10)**
Line feed	**Chr(10)**

When writing to text files, BasicScript uses the end-of-line appropriate to that platform. You can retrieve the same end-of-line used by BasicScript using the **Basic.Eoln$** property:

```
crlf = Basic.Eoln$
Print #1,"Line 1." & crlf & "Line 2."
```

Alignment

A major difference between platforms supported by BasicScript is the forced alignment of data. BasicScript handles most alignment issues itself.

Portability of Compiled Code

Scripts compiled under BasicScript can be executed without recompilation on any platform supported by BasicScript.

Unsupported Language Elements

A compiled BasicScript script is portable to any platform on which BasicScript runs. Because of this, it is possible to execute a script that was compiled on another platform and contains calls to language elements not supported by the current platform.

BasicScript generates a runtime error when unsupported language elements are encountered during execution. For example, the following script will execute without errors under Windows but generate a runtime error when run under UNIX:

```
Sub Main()
   MsgBox "Hello, world."
End Sub
```

If you trap a call to an unsupported function, the function will return one of the following values:

Data Type	Skipped Function Returns
Integer	0
Double	0.0
Single	0.0
Long	0
Date	December 31, 1899
Boolean	**False**
Variant	Empty
Object	Nothing

Path Separators

Different file systems use different characters to separate parts of a pathname. For example, under Windows, Win32, and OS/2, the backslash character is used:

```
s$ = "c:\sheets\bob.xls"
```

Under UNIX, the forward slash is used:

```
s$ = "/sheets/bob.xls"
```

When creating scripts that operate on any of these platforms, BasicScript recognizes the forward slash universally as a valid path separator. Thus, the following file specification is valid on all these platforms:

```
s$ = "/sheets/bob.xls"
```

On the Macintosh, the slashes are valid filename characters. Instead, BasicScript recognizes the colon as the valid file separator character:

```
s$ = "sheets:bob.xls"
```

You can find out the path separator character for your platform using the **Basic.PathSeparator$** property:

```
s$ = "sheets" & Basic.PathSeparator$ & "bob.xls"
```

Relative Paths

Specifying relative paths is different across platforms. Under UNIX, Windows, Win32, and OS/2, a period (.) is used to specify the current directory, and two periods (..) are used to indicate the parent directory, as shown below:

```
s$ = ".\bob.xls"     'File in the current directory
s$ = "..\bob.xls"    'File in the parent directory
```

On the Macintosh, double colons are used to specify the parent folder:

```
s$ = "::bob.xls"'File in the parent folder
```

Drive Letters

Not all platforms support drive letters. For example, considering the following file specification:

```
c:\test.txt
```

Under UNIX, this specifies a single file called c:\test.txt. Under Windows, this specifies a file called test.txt in the root directory of drive c. On the Macintosh, this specifies a file called \test.txt in a folder called c. You can use the **Basic.Capability** method to determine whether your platform supports drive letters:

```
Sub Main()
   If Basic.Capability(1) Then s$ = "c:/" Else s$ = ""
   s$ = s$ & "test.xls"
   MsgBox "The platform-specific filename is: " & s$
End Sub
```

UNC Pathnames

Many platforms support UNC pathnames, including Windows and Win32. If you choose to use these, make sure that UNC pathnames are supported on the platforms on which your script will run.

CSng (function)

Syntax `CSng(`*expression*`)`

Description Converts *expression* to a **Single**.

Comments This function accepts any expression convertible to a **Single**, including strings. A runtime error is generated if *expression* is **Null**. **Empty** is treated as 0.0.

A runtime error results if the passed expression is not within the valid range for **Single**.

When passed a numeric expression, this function has the same effect as assigning the numeric expression to a **Single**.

When used with variants, this function guarantees that the expression is converted to a **Single** variant (**VarType** 4).

Example

```
'This example displays the value of a String converted to a
'Single.
Sub Main()
   s$ = "100"
   MsgBox "The single value is: " & CSng(s$)
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CStr** (function); **CVar** (function); **CVErr** (function); **Single** (data type).

Platform(s) All.

CStr (function)

Syntax `CStr(`*expression*`)`

Description Converts *expression* to a **String**.

Comments Unlike **Str$** or **Str**, the string returned by **CStr** will not contain a leading space if the expression is positive. Further, the **CStr** function correctly recognizes thousands and decimal separators for your locale.

Different data types are converted to **String** in accordance with the following rules:

Data Type	CStr Returns
Any numeric type	A string containing the number without the leading space for positive values
Date	A string converted to a date using the short date format
Boolean	A string containing either "True" or "False"
Null variant	A runtime error
Empty variant	A zero-length string

Example

```
'This example displays the value of a Double converted to a
'String.
Sub Main()
   s# = 123.456
   MsgBox "The string value is: " & CStr(s#)
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CVar** (function); **CVErr** (function); **String** (data type); **Str**, **Str$** (functions).

Platform(s) All.

CurDir, CurDir$ (functions)

Syntax `CurDir[$][(`*drive*`)]`

Description Returns the current directory on the specified drive. If no *drive* is specified or *drive* is zero-length, then the current directory on the current drive is returned.

Comments **CurDir$** returns a **String**, whereas **CurDir** returns a **String** variant.

BasicScript generates a runtime error if *drive* is invalid.

Example

```
'This example saves the current directory, changes to the next
'higher directory, and displays the change; then restores the
'original directory and displays the change. Note: The dot
'designators will not work with all platforms.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   save$ = CurDir$
   ChDir (".")
   MsgBox "Old directory: " & save$ & crlf & "New directory: " _
      & CurDir$
   ChDir (save$)
   MsgBox "Directory restored to: " & CurDir$
End Sub
```

See Also **ChDir** (statement); **ChDrive** (statement); **Dir**, **Dir$** (functions); **MkDir** (statement); **RmDir** (statement).

Platform(s) All.

Platform Notes **UNIX:** On UNIX platforms, the *drive* parameter is ignored. Since UNIX platforms do not support drive letters, the current directory is always returned.

NetWare: Since NetWare does not support drive letters, the *drive* parameter specifies a volume name (up to 14 characters). The returned value will have the following format:

```
volume:[dir[\dir]...]
```

Currency (data type)

Syntax `Currency`

Description A data type used to declare variables capable of holding fixed-point numbers with 15 digits to the left of the decimal point and 4 digits to the right.

Comments **Currency** variables are used to hold numbers within the following range:

```
-922,337,203,685,477.5808 <= currency <=
922,337,203,685,477.5807
```

Due to their accuracy, **Currency** variables are useful within calculations involving money.

The type-declaration character for **Currency** is @.

Storage

Internally, currency values are 8-byte integers scaled by 10000. Thus, when appearing within a structure, currency values require 8 bytes of storage. When used with binary or random files, 8 bytes of storage are required.

See Also **Date** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CCur** (function).

Platform(s) All.

CVar (function)

Syntax `CVar(`*expression*`)`

Description Converts *expression* to a **Variant**.

Comments This function is used to convert an expression into a variant. Use of this function is not necessary (except for code documentation purposes) because assignment to variant variables automatically performs the necessary conversion:

```
Sub Main()
   Dim v As Variant
   v = 4 & "th"              'Assigns "4th" to v.
   MsgBox "You came in: " & v
   v = CVar(4 & "th")        'Assigns "4th" to v.
   MsgBox "You came in: " & v
End Sub
```

Example

```
'This example converts an expression into a Variant.
Sub Main()
   Dim s As String
   Dim a As Variant
   s = CStr("The quick brown fox ")
   message = CVar(s & "jumped over the lazy dog.")
   MsgBox message
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVErr** (function); **Variant** (data type).

Platform(s) All.

CVErr (function)

Syntax `CVErr(`*expression*`)`

Description Converts *expression* to an error.

Comments This function is used to convert an expression into a user-defined error number.

A runtime error is generated under the following conditions:

- If *expression* is **Null**.
- If *expression* is a number outside the legal range for errors, which is as follows:
  `0 <=` *expression* `<= 65535`
- If *expression* is Boolean.
- If *expression* is a **String** that can't be converted to a number within the legal range.

Empty is treated as 0.

Example

```
'This example simulates a user-defined error and displays the
'error number.
Sub Main()
   MsgBox "The error is: " & CStr(CVErr(2046))
End Sub
```

See Also **CCur** (function); **CBool** (function); **CDate**, **CVDate** (functions); **CDbl** (function); **CInt** (function); **CLng** (function); **CSng** (function); **CStr** (function); **CVar** (function), **IsError** (function).

Platform(s) All.

Date (data type)

Syntax `Date`

Description A data type capable of holding date and time values.

Comments **Date** variables are used to hold dates within the following range:

```
January 1, 100 00:00:00 <= date <= December 31, 9999 23:59:59
-6574340 <= date <= 2958465.99998843
```

Internally, dates are stored as 8-byte IEEE double values. The integer part holds the number of days since December 31, 1899, and the fractional part holds the number of seconds as a fraction of the day. For example, the number 32874.5 represents January 1, 1990 at 12:00:00.

When appearing within a structure, dates require 8 bytes of storage. Similarly, when used with binary or random files, 8 bytes of storage are required.

There is no type-declaration character for **Date**.

Date variables that haven't been assigned are given an initial value of 0 (i.e., December 31, 1899).

Date Literals

Literal dates are specified using number signs, as shown below:

```
Dim d As Date
d = #January 1, 1990#
```

The interpretation of the date string (i.e., January 1, 1990 in the above example) occurs at runtime, using the current country settings. This is a problem when interpreting dates such as 1/2/1990. If the date format is M/D/Y, then this date is January 2, 1990. If the date format is D/M/Y, then this date is February 1, 1990. To remove any ambiguity when interpreting dates, use the universal date format:

date_variable `= #YY/MM/DD HH:MM:SS#`

The following example specifies the date June 3, 1965, using the universal date format:

```
Dim d As Date
d = #1965/6/3 10:23:45#
```

See Also **Currency** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CDate**, **CVDate** (functions).

Platform(s) All.

Date, Date$ (functions)

Syntax `Date[$][()]`

Description Returns the current system date.

Comments The **Date$** function returns the date using the short date format. The **Date** function returns the date as a **Date** variant.

Use the **Date/Date$** statements to set the system date.

Note: In prior versions of BasicScript, the **Date$** function returned the date using a fixed date format. The date is now returned using the current short date format (defined by the operating system), which may differ from the previous fixed format.

Example

```
'This example saves the current date to Cdate$, then changes
'the date and displays the result. It then changes the date
'back to the saved date and displays the result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   TheDate$ = Date$()
   Date$ = "01/01/95"
   MsgBox "Saved date is: " & TheDate$ & crlf _
      & "Changed date is: " & Date$()
   Date$ = TheDate$
   MsgBox "Restored date to: " & TheDate$
End Sub
```

See Also **CDate**, **CVDate** (functions); **Time**, **Time$** (functions); **Date**, **Date$** (statements); **Now** (function); **Format**, **Format$** (functions); **DateSerial** (function); **DateValue** (function).

Platform(s) All.

Date, Date$ (statements)

Syntax `Date[$] =` *newdate*

Description Sets the system date to the specified date.

Comments The **Date$** statement requires a string variable using one of the following formats:

MM-DD-YYYY

MM-DD-YY

MM/DD/YYYY

MM/DD/YY,

where *MM* is a two-digit month between 1 and 31, *DD* is a two-digit day between 1 and 31, and *YYYY* is a four-digit year between 1/1/100 and 12/31/9999.

The **Date** statement converts any expression to a date, including string and numeric values. Unlike the **Date$** statement, **Date** recognizes many different date formats, including abbreviated and full month names and a variety of ordering options. If *newdate* contains a time component, it is accepted, but the time is not changed. An error occurs if *newdate* cannot be interpreted as a valid date.

Example

```
'This example saves the current date to Cdate$, then changes
'the date and displays the result. It then changes the date
'back to the saved date and displays the result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   TheDate$ = Date$()
   Date$ = "01/01/95"
   MsgBox "Saved date is: " & TheDate$ & crlf _
      & "Changed date is: " & Date$()
   Date$ = TheDate$
   MsgBox "Restored date to: " & TheDate$
End Sub
```

See Also **Date**, **Date$** (functions); **Time**, **Time$** (statements).

Platform(s) All.

Platform Notes On some platforms, you may not have permission to change the date, causing runtime error 70 to be generated. This can occur on all UNIX platforms, Win32, and OS/2.

The range of valid dates varies from platform to platform. The following table describes the minimum and maximum dates accepted by various platforms:

Platform	Minimum Date	Maximum Date
Macintosh	January 1, 1904	February 6, 2040
Windows	January 1, 1980	December 31, 2099
Windows 95	January 1, 1980	December 31, 2099
OS/2	January 1, 1980	December 31, 2079
NetWare	January 1, 1980	December 31, 2099

DateAdd (function)

Syntax `DateAdd(`*interval, number, date*`)`

Description Returns a **Date** variant representing the sum of date and a specified number (*number*) of time intervals *(interval*).

Comments

This function adds a specified number (*number*) of time intervals (*interval*) to the specified date (*date*). The following table describes the named parameters to the **DateAdd** function:

Named Parameter	Description
interval	**String** expression indicating the time interval used in the addition.
number	**Integer** indicating the number of time intervals you wish to add. Positive values result in dates in the future; negative values result in dates in the past.
date	Any expression convertible to a **Date** string expression. An example of a valid date/time string would be "January 1, 1993".

The *interval* parameter specifies what unit of time is to be added to the given date. It can be any of the following:

Time	Interval
"y"	Day of the year
"yyyy"	Year
"d"	Day
"m"	Month
"q"	Quarter
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
"w"	Weekday

To add days to a date, you may use either day, day of the year, or weekday, as they are all equivalent ("d", "y", "w").

The **DateAdd** function will never return an invalid date/time expression. The following example adds two months to December 31, 1992:

```
s# = DateAdd("m", 2, "December 31, 1992")
```

In this example, s is returned as the double-precision number equal to "February 28, 1993", not "February 31, 1993".

BasicScript generates a runtime error if you try subtracting a time interval that is larger than the time value of the date.

Example

```
'This example gets today's date using the Date$ function; adds
'three years, two months, one week, and two days to it; and
```

```
'then displays the result in a dialog box.
Sub Main()
   Dim sdate$
   sdate$ = Date$
   NewDate# = DateAdd("yyyy", 4, sdate$)
   NewDate# = DateAdd("m", 3, NewDate#)
   NewDate# = DateAdd("ww", 2, NewDate#)
   NewDate# = DateAdd("d", 1, NewDate#)
   s$ = "Four years, three months, two weeks,"
   s$ = s$ + " and one day from now will be: "
   s$ = s$ & Format(NewDate#, "long date")
   MsgBox s$
End Sub
```

See Also **DateDiff** (function).

Platform(s) All.

DateDiff (function)

Syntax `DateDiff(`*interval*`, `*date1*`, `*date2*` [, [`*firstdayofweek*`] [,`*firstweekofyear*`]])`

Description Returns a **Date** variant representing the number of given time intervals between *date1* and *date2*.

Comments The following describes the named parameters:

Named Parameter	Description
interval	**String** expression indicating the specific time interval you wish to find the difference between. An error is generated if *interval* is **Null**.
date1	Any expression convertible to a **Date**. An example of a valid date/time string would be "January 1, 1994".
date2	Any expression convertible to a **Date**. An example of a valid date/time string would be "January 1, 1994".
firstdayofweek	Indicates the first day of the week. If omitted, then sunday is assumed (i.e., the constant **ebSunday** described below).
firstweekofyear	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant **ebFirstJan1** as described bellow).

The following lists the valid time interval strings and the meanings of each. The **Format$** function uses the same expressions.

Time	Interval
"y"	Day of the year
"yyyy"	Year
"d"	Day
"m"	Month
"q"	Quarter
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
"w"	Weekday

To find the number of days between two dates, you may use either day or day of the year, as they are both equivalent ("d", "y").

The time interval weekday ("w") will return the number of weekdays occurring between *date1* and *date2*, counting the first occurrence but not the last. However, if the time interval is week ("ww"), the function will return the number of calendar weeks between *date1* and *date2*, counting the number of Sundays. If *date1* falls on a Sunday, then that day is counted, but if *date2* falls on a Sunday, it is not counted.

The *firstdayofweek* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofweek*.
ebSunday	1	Sunday (the default)
ebMonday	2	Monday
ebTuesday	3	Tuesday
ebWednesday	4	Wednesday
ebThursday	5	Thursday
ebFriday	6	Friday
ebSaturday	7	Saturday

The *firstdayofyear* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofyear*.

Constant	Value	Description
ebFirstJan1	1	The first week of the year is that in which January 1 occurs (the default).
ebFirstFourDays	2	The first week of the year is that containing at least four days in the year.
ebFirstFullWeek	3	The first week of the year is the first full week of the year.

The **DateDiff** function will return a negative date/time value if *date1* is a date later in time than *date2*. If *date1* or *date2* are **Null**, then **Null** is returned.

Example

```
'This example gets today's date and adds ten days to it. It
'then calculates the difference between the two dates in days
'and weeks and displays the result.
Sub Main()
   today$ = Format(Date$,"Short Date")
   NextWeek = Format(DateAdd("d", 14, today$),"Short Date")
   DifDays# = DateDiff("d", today$, NextWeek)
   DifWeek# = DateDiff("w", today$, NextWeek)
   s$ = "The difference between " & today$ & " and " & NextWeek
   s$ = s$ & " is: " & DifDays# & " days or " _
      & DifWeek# & " weeks"
   MsgBox s$
End Sub
```

See Also **DateAdd** (function).

Platform(s) All.

DatePart (function)

Syntax `DatePart(`*interval*`,` *date* `[, [`*firstdayofweek*`] [,`*firstweekofyear*`]])`

Description Returns an **Integer** representing a specific part of a date/time expression.

Comments The **DatePart** function decomposes the specified date and returns a given date/time element. The following table describes the named parameters:

Named Parameter	Description
interval	**String** expression that indicates the specific time interval you wish to identify within the given date.
date	Any expression convertible to a **Date**. An example of a valid date/time string would be "January 1, 1995".
firstdayofweek	Indicates the first day of the week. If omitted, then sunday is assumed (i.e., the constant **ebSunday** described below).

Named Parameter	Description
firstweekofyear	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant **ebFirstJan1** as described bellow).

The following table lists the valid time interval strings and the meanings of each. The **Format$** function uses the same expressions.

Time	Interval
"y"	Day of the year
"yyyy"	Year
"d"	Day
"m"	Month
"q"	Quarter
"ww"	Week
"h"	Hour
"n"	Minute
"s"	Second
"w"	Weekday

The *firstdayofweek* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofweek.*
ebSunday	1	Sunday (the default)
ebMonday	2	Monday
ebTuesday	3	Tuesday
ebWednesday	4	Wednesday
ebThursday	5	Thursday
ebFriday	6	Friday
ebSaturday	7	Saturday

The *firstdayofyear* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofyear*.
ebFirstJan1	1	The first week of the year is that in which January 1 occurs (the default).

Constant	Value	Description
ebFirstFourDays	2	The first week of the year is that containing at least four days in the year.
ebFirstFullWeek	3	The first week of the year is the first full week of the year.

Example

```
'This example displays the parts of the current date.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   today$ = Date$
   qtr = DatePart("q",today$)
   yr = DatePart("yyyy",today$)
   mo = DatePart("m",today$)
   wk = DatePart("ww",today$)
   da = DatePart("d",today$)
   s$ = "Quarter: " & qtr & crlf
   s$ = s$ & "Year: " & yr & crlf
   s$ = s$ & "Month: " & mo & crlf
   s$ = s$ & "Week: " & wk & crlf
   s$ = s$ & "Day: " & da & crlf
   MsgBox s$
End Sub
```

See Also **Day** (function); **Minute** (function); **Second** (function); **Month** (function); **Year** (function); **Hour** (function); **Weekday** (function); **Format, Format$** (functions).

Platform(s) All.

DateSerial (function)

Syntax `DateSerial(`*year*`, `*month*`, `*day*`)`

Description Returns a **Date** variant representing the specified date.

Comments The **DateSerial** function takes the following named parameters:

Named Parameter	Description
year	**Integer** between 100 and 9999
month	**Integer** between 1 and 12
day	**Integer** between 1 and 31

Example

```
'This example converts a date to a real number representing the
'serial date in days since December 30, 1899 (which is day 0).
Sub Main()
   tdate# = DateSerial(1993,08,22)
```

```
   MsgBox "The DateSerial value for August 22, 1993, is: " _
      & tdate#
End Sub
```

See Also **DateValue** (function); **TimeSerial** (function); **TimeValue** (function); **CDate**, **CVDate** (functions).

Platform(s) All.

DateValue (function)

Syntax `DateValue(`*date*`)`

Description Returns a **Date** variant representing the date contained in the specified string argument.

Example

```
'This example returns the day of the month for today's date.
Sub Main()
   tdate$ = Date$
   tday = DateValue(tdate$)
   MsgBox tdate & " date value is: " & tday$
End Sub
```

See Also **TimeSerial** (function); **TimeValue** (function); **DateSerial** (function).

Platform(s) All.

Platform Notes **Windows:** Under Windows, date specifications vary depending on the international settings contained in the "intl" section of the win.ini file. The date items must follow the ordering determined by the current date format settings in use by Windows..

Day (function)

Syntax `Day(`*date*`)`

Description Returns the day of the month specified by *date*.

Comments The value returned is an **Integer** between 0 and 31 inclusive.

The *date* parameter is any expression that converts to a **Date**.

Example

```
'This example gets the current date and then displays it.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   CurDate = Now()
   MsgBox "Today is day " & Day(CurDate) & " of the month." _
      & crlf &  "Tomorrow is day " & Day(CurDate + 1)
End Sub
```

See Also **Minute** (function); **Second** (function); **Month** (function); **Year** (function); **Hour** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

DDB (function)

Syntax `DDB(`*cost*`, `*salvage*`, `*life*`, `*period*` [,`*factor*`])`

Description Calculates the depreciation of an asset for a specified *period* of time using the double-declining balance method.

Comments The double-declining balance method calculates the depreciation of an asset at an accelerated rate. The depreciation is at its highest in the first period and becomes progressively lower in each additional period. **DDB** uses the following formula to calculate the depreciation:

`DDB =((`*Cost-Total_depreciation_from_all_other_periods*`) * 2)/`*Life*

The **DDB** function uses the following named parameters:

Named Parameter	Description
cost	**Double** representing the initial cost of the asset
salvage	**Double** representing the estimated value of the asset at the end of its predicted useful life
life	**Double** representing the predicted length of the asset's useful life
period	**Double** representing the period for which you wish to calculate the depreciation
factor	Depreciation factor determining the rate the balance declines. If this parameter is missing, then 2 is assumed (double-declining method).

The *life* and *period* parameters must be expressed using the same units. For example, if *life* is expressed in months, then *period* must also be expressed in months.

Example

```
'This example calculates the depreciation for capital equipment
'that cost $10,000, has a service life of ten years, and is
'worth $2,000 as scrap. The dialog box displays the depreciation
'for each of the first four years.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   s$ = "Depreciation Table" & crlf & crlf
   For yy = 1 To 4
      CurDep# = DDB(10000.0,2000.0,10,yy)
      s$ = s$ & "Year " & yy & " : " & CurDep# & crlf
```

```
   Next yy
   MsgBox s$
End Sub
```

See Also **Sln** (function); **SYD** (function).

Platform(s) All.

DDEExecute (statement)

Syntax `DDEExecute` *channel, command$*

Description Executes a command in another application.

Comments The **DDEExecute** statement takes the following parameters:

Parameter	Description
channel	**Integer** containing the DDE channel number returned from **DDEInitiate**. An error will result if *channel* is invalid.
command$	**String** containing the command to be executed. The format of *command$* depends on the receiving application.

If the receiving application does not execute the instructions, BasicScript generates a runtime error.

Example

```
'This example selects a cell in an Excel spreadsheet.
Sub Main()
   q$ = Chr(34)
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   cmd$ = "Select(" & q$ & "R1C1:R8C1" & q$ & ")"
   DDEExecute ch%,cmd$
   DDETerminate ch%
End Sub
```

See Also **DDEInitiate** (function); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDESend** (statement); **DDETerminate** (statement); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDEInitiate (function)

Syntax `DDEInitiate(`*application$, topic$*`)`

Description Initializes a DDE link to another application and returns a unique number subsequently used to refer to the open DDE channel.

Comments The **DDEInitiate** statement takes the following parameters:

Parameter	Description
application$	**String** containing the name of the application (the server) with which a DDE conversation will be established.
topic$	**String** containing the name of the topic for the conversation. The possible values for this parameter are described in the documentation for the server application.

This function returns 0 if BasicScript cannot establish the link. This will occur under any of the following circumstances:

- The specified application is not running.
- The topic was invalid for that application.
- Memory or system resources are insufficient to establish the DDE link.

Example

```
'This example selects a range of cells in an Excel spreadsheet.
Sub Main()
   q$ = Chr(34)
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   cmd$ = "Select(" & q$ & "R1C1:R8C1" & q$ & ")"
   DDEExecute ch%,cmd$
   DDETerminate ch%
End Sub
```

See Also **DDEExecute** (statement); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDESend** (function); **DDETerminate** (statement); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDEPoke (statement)

Syntax `DDEPoke` *channel*, *DataItem*, *value*

Description Sets the value of a data item in the receiving application associated with an open DDE link.

Comments The **DDEPoke** statement takes the following parameters:

Parameter	Description
channel	**Integer** containing the DDE channel number returned from DDEInitiate. An error will result if *channel* is invalid.
DataItem	Data item to be set. This parameter can be any expression convertible to a **String**. The format depends on the server.
value	The new value for the data item. This parameter can be any expression convertible to a **String**. The format depends on the server. A runtime error is generated if *value* is **Null**.

Example

```
'This example pokes a value into an Excel spreadsheet.
Sub Main()
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   DDEPoke ch%,"R1C1","980"
   DDETerminate ch%
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDERequest**, **DDERequest$** (functions); **DDESend** (function); **DDETerminate** (statement); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDERequest, DDERequest$ (functions)

Syntax `DDERequest[$](`*channel*`,`*DataItem$*`)`

Description Returns the value of the given data item in the receiving application associated with the open DDE channel.

Comments **DDERequest$** returns a **String**, whereas **DDERequest** returns a **String** variant.

The **DDERequest/DDERequest$** functions take the following parameters:

Parameter	Description
channel	**Integer** containing the DDE channel number returned from **DDEInitiate**. An error will result if *channel* is invalid.
DataItem$	**String** containing the name of the data item to request. The format for this parameter depends on the server.

The format for the returned value depends on the server.

Example

```
'This example gets a value from an Excel spreadsheet.
Sub Main()
   ch% = DDEInitiate("Excel","c:\excel\test.xls")
   s$ = DDERequest$(ch%,"R1C1")
   DDETerminate ch%
   MsgBox s$
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDEPoke** (statement); **DDESend** (function); **DDETerminate** (statement); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDESend (statement)

Syntax `DDESend` *application$, topic$, DataItem, value*

Description Initiates a DDE conversation with the server as specified by *application$* and *topic$* and sends that server a new value for the specified item.

Comments The **DDESend** statement takes the following parameters:

Parameter	Description
application$	**String** containing the name of the application (the server) with which a DDE conversation will be established.
topic$	**String** containing the name of the topic for the conversation. The possible values for this parameter are described in the documentation for the server application.
DataItem	Data item to be set. This parameter can be any expression convertible to a **String**. The format depends on the server.

Parameter	Description
value	New value for the data item. This parameter can be any expression convertible to a **String**. The format depends on the server. A runtime error is generated if *value* is **Null**.

The **DDESend** statement performs the equivalent of the following statements:

```
ch% = DDEInitiate(application$, topic$)
DDEPoke ch%, item, data
DDETerminate ch%
```

Example

```
'This code fragment sets the content of the first cell in an
'Excel spreadsheet.
Sub Main()
   On Error Goto Trap1
   DDESend "Excel","c:\excel\test.xls","R1C1","Hello, world."
   On Error Goto 0
   'Add more lines here.
Trap1:
   MsgBox "Error sending data to Excel."
   Exit Sub'Reset error handler.
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDETerminate** (statement); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDETerminate (statement)

Syntax `DDETerminate` *channel*

Description Closes the specified DDE channel.

Comments The *channel* parameter is an **Integer** containing the DDE channel number returned from **DDEInitiate**. An error will result if *channel* is invalid.

All open DDE channels are automatically terminated when the script ends.

Example

```
'This code fragment sets the content of the first cell in an
'Excel spreadsheet.
Sub Main()
```

```
   q$ = Chr(34)
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   cmd$ = "Select(" & q$ & "R1C1:R8C1" & q$ & ")"
   DDEExecute ch%,cmd$
   DDETerminate ch%
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDESend** (function); **DDETerminateAll** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDETerminateAll (statement)

Syntax `DDETerminateAll`

Description Closes all open DDE channels.

Comments All open DDE channels are automatically terminated when the script ends.

Example

```
'This code fragment selects the contents of the first cell in
'an Excel spreadsheet.
Sub Main()
   q$ = Chr(34)
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   cmd$ = "Select(" & q$ & "R1C1:R8C1" & q$ & ")"
   DDEExecute ch%,cmd$
   DDETerminateAll
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDESend** (function); **DDETerminate** (statement); **DDETimeout** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

DDETimeout (statement)

Syntax `DDETimeout` *milliseconds*

Description Sets the number of milliseconds that must elapse before any DDE command times out.

Comments The *milliseconds* parameter is a **Long** and must be within the following range:

```
0 <= milliseconds <= 2,147,483,647
```

The default is 10,000 (10 seconds).

Example

```
Sub Main()
   q$ = Chr(34)
   ch% = DDEInitiate("Excel","c:\sheets\test.xls")
   DDETimeout(20000)
   cmd$ = "Select(" & q$ & "R1C1:R8C1" & q$ & ")"
   DDEExecute ch%,cmd$
   DDETerminate ch%
End Sub
```

See Also **DDEExecute** (statement); **DDEInitiate** (function); **DDEPoke** (statement); **DDERequest**, **DDERequest$** (functions); **DDESend** (function); **DDETerminate** (statement); **DDETerminateAll** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows:** Under Windows, the DDEML library is required for DDE support. This library is loaded when the first **DDEInitiate** statement is encountered and remains loaded until the BasicScript system is terminated. Thus, the DDEML library is required only if DDE statements are used within a script.

Declare (statement)

Syntax `Declare {Sub | Function}` *name*`[`*TypeChar*`] [CDecl | Pascal | System | StdCall] [Lib "`*LibName$*`" [Alias "`*AliasName$*`"]] [([`*ParameterList*`])] [As` *type*`]`

Where *ParameterList* is a comma-separated list of the following (up to 30 parameters are allowed):

`[Optional] [ByVal | ByRef]` *ParameterName*`[()] [As` *ParameterType*`]`

Description Creates a prototype for either an external routine or a BasicScript routine that occurs later in the source module or in another source module.

Comments **Declare** statements must appear outside of any **Sub** or **Function** declaration.

Declare statements are only valid during the life of the script in which they appear.

The **Declare** statement uses the following parameters:

Parameter	Description
name	Any valid BasicScript name. When you declare functions, you can include a type-declaration character to indicate the return type. This name is specified as a normal BasicScript keyword—i.e., it does not appear within quotes.
TypeChar	An optional type-declaration character used when defining the type of data returned from functions. It can be any of the following characters: #, !, $, @, %, or &. For external functions, the @ character is not allowed. Type-declaration characters can only appear with function declarations, and take the place of the **As** *type* clause. **Note: Currency** data cannot be returned from external functions. Thus, the @ type-declaration character cannot be used when declaring external functions.
Decl	Optional keyword indicating that the external subroutine or function uses the C calling convention. With C routines, arguments are pushed right to left on the stack and the caller performs stack cleanup.
Pascal	Optional keyword indicating that this external subroutine or function uses the Pascal calling convention. With Pascal routines, arguments are pushed left to right on the stack and the called function performs stack cleanup.
System	Optional keyword indicating that the external subroutine or function uses the System calling convention. With System routines, arguments are pushed right to left on the stack, the caller performs stack cleanup, and the number of arguments is specified in the **AL** register.
StdCall	Optional keyword indicating that the external subroutine or function uses the StdCall calling convention. With StdCall routines, arguments are pushed right to left on the stack and the called function performs stack cleanup.
LibName$	Must be specified if the routine is external. This parameter specifies the name of the library or code resource containing the external routine and must appear within quotes. The *LibName$* parameter can include an optional path specifying the exact location of the library or code resource.

Parameter	Description

AliasName$	Alias name that must be given to provide the name of the routine if the *name* parameter is not the routine's real name. For example, the following two statements declare the same routine:
	`Declare Function GetCurrentTime Lib "user" () As Integer`
	`Declare Function GetTime Lib "user" Alias "GetCurrentTime" _`
	`  As Integer`
	Use an alias when the name of an external routine conflicts with the name of a BasicScript internal routine or when the external routine name contains invalid characters.
	The *AliasName$* parameter must appear within quotes.

type	Indicates the return type for functions. For external functions, the valid return types are: **Integer**, **Long**, **String**, **Single**, **Double**, **Date**, **Boolean**, and data objects. **Note: Currency**, **Variant**, fixed-length strings, arrays, user-defined types, and OLE Automation objects cannot be returned by external functions.
Optional	Keyword indicating that the parameter is optional. All optional parameters must be of type **Variant**. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter being defined is required when calling this subroutine or function.
ByVal	Optional keyword indicating that the caller will pass the parameter by value. Parameters passed by value cannot be changed by the called routine.
ByRef	Optional keyword indicating that the caller will pass the parameter by reference. Parameters passed by reference can be changed by the called routine. If neither **ByVal** or **ByRef** are specified, then **ByRef** is assumed.

Parameter	Description
ParameterName	Name of the parameter, which must follow BasicScript naming conventions: 1. Must start with a letter. 2. May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. 3. Must not exceed 80 characters in length. Additionally, *ParameterName* can end with an optional type-declaration character specifying the type of that parameter (i.e., any of the following characters: %, &, !, #, @).
()	Indicates that the parameter is an array.
ParameterType	Specifies the type of the parameter (e.g., **Integer, String**, **Variant**, and so on). The **As** *ParameterType* clause should only be included if *ParameterName* does not contain a type-declaraction character. In addition to the default BasicScript data types, *ParameterType* can specify any user-defined structure, data object, or OLE Automation object. If the data type of the parameter is not known in advance, then the **Any** keyword can be used. This forces the BasicScript compiler to relax type checking, allowing any data type to be passed in place of the given argument. `Declare Sub Convert Lib "mylib" (a As Any)` The **Any** data type can only be used when passing parameters to external routines.

Passing Parameters

By default, BasicScript passes arguments by reference. Many external routines require a value rather than a reference to a value. The **ByVal** keyword does this. For example, this C routine

```
void MessageBeep(int);
```

would be declared as follows:

```
Declare Sub MessageBeep Lib "user" (ByVal n As Integer)
```

As an example of passing parameters by reference, consider the following C routine which requires a pointer to an integer as the third parameter:

```
int SystemParametersInfo(int,int,int *,int);
```

This routine would be declared as follows (notice the **ByRef** keyword in the third parameter):

```
Declare Function SystemParametersInfo Lib "user" (ByVal _
   action As Integer, ByVal uParam As Integer,ByRef pInfo _
   As Integer, ByVal updateINI As Integer) As Integer
```

Strings can be passed by reference or by value. When they are passed by reference, a pointer to a pointer to a null-terminated string is passed. When they are passed by value, BasicScript passes a pointer to a null-terminated string (i.e., a C string).

When passing a string by reference, the external routine can change the pointer or modify the contents of the existing. If an external routine modifies a passed string variable (regardless of whether the string was passed by reference or by value), then there must be sufficient space within the string to hold the returned characters. This can be accomplished using the **Space** function, as shown in the following example which calls a Windows 16-bit DLL:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal _
   dirname$, ByVal length%)
Sub Main()
   Dim s As String
   s = Space(128)
   GetWindowsDirectory s,128
End Sub
```

Another alternative to ensure that a string has sufficient space is to declare the string with a fixed length:

```
Declare Sub GetWindowsDirectory Lib "kernel" (ByVal _
   dirname$, ByVal length%)
Sub Main
   Dim s As String * 128
   GetWindowsDirectory s,len(s)
End Sub
```

Calling Conventions with External Routines

For external routines, the argument list must exactly match that of the referenced routine. When calling an external subroutine or function, BasicScript needs to be told how that routine expects to receive its parameters and who is responsible for cleanup of the stack.

The following table describes BasicScript's calling conventions and how these translate to those supported by C.

BasicScript Calling Convention	C Calling Convention	Characteristics
StdCall	**_stdcall**	Arguments are pushed right to left.
		The called function performs stack cleanup.
Pascal	**pascal**	Arguments are pushed left to right.
		The called function performs stack cleanup
System	**_System**	Arguments are pushed right to left.
		The caller performs stack cleanup.
		The number of arguments is specified in the ax 1 register.
CDecl	**cdec1**	Arguments are pushed right to left.
		The caller performs stack cleanup.

The following table shows which calling conventions are supported on which platform, and indicates what the default calling convention is when no explicit calling convention is specified in the **Declare** statement.

Platform	Supported Calling Conventions	Default Calling Convention
Windows	**Pascal, CDecl**	**Pascal**
Win32/Intel	**Pascal, CDecl, StdCall**	**StdCall**
Win32/PPC	**CDecl**	**CDecl**
Macintosh	On the 68K, the Macintosh supports only the **CDecl** calling convention.	On the 68K, the default calling convention is **CDecl**.
	The PowerMac supports a single calling convention that evaluates parameters left to right. No special calling convention keywords are required.	On the 68K, a runtime error occurs if any explicit calling convention keyword is specified.
OS/2	**System, Pascal, CDecl**	**System**
NetWare	**CDecl, Pascal**	**CDecl**
UNIX	**CDecl**	**CDecl**

Passing Null Pointers

For external routines defined to receive strings by value, BasicScript passes uninitialized strings as null pointers (a pointer whose value is 0). The constant **ebNullString** can be used to force a null pointer to be passed as shown below:

```
Declare Sub Foo Lib "sample" (ByVal lpName As Any)
Sub Main()
   Foo ebNullString     'pass a null pointer
End Sub
```

Another way to pass a null pointer is to declare the parameter that is to receive the null pointer as type **Any**, then pass a long value 0 by value:

```
Declare Sub Foo Lib "sample" (ByVal lpName As Any)
Sub Main()
   Foo ByVal 0&         'Pass a null pointer.
End Sub
```

Passing Data to External Routines

The following table shows how the different data types are passed to external routines:

Data type	Is passed as
ByRef Boolean	A pointer to a 2-byte value containing –1 or 0.
ByVal Boolean	A 2-byte value containing –1 or 0.
ByVal Integer	A pointer to a 2-byte short integer.
ByRef Integer	A 2-byte short integer.
ByVal Long	A pointer to a 4-byte long integer.
ByRef Long	A 4-byte long integer.
ByRef Single	A pointer to a 4-byte IEEE floating-point value (a float).
ByVal Single	A 4-byte IEEE floating-point value (a float).
ByRef Double	A pointer to an 8-byte IEEE floating-point value (a double).
ByVal Double	An 8-byte IEEE floating-point value (a double).
ByVal String	A pointer to a null-terminated string. With strings containing embedded nulls (**Chr$**(0)), it is not possible to determine which null represents the end of the string; therefore, the first null is considered the string terminator. An external routine can freely change the content of a string. It cannot, however, write beyond the end of the null terminator.

Data type	Is passed as
ByRef String	A pointer to a pointer to a null-terminated string. With strings containing embedded nulls (**Chr$**(0)), it is not possible to determine which null represents the end of the string; therefore, the first null is considered the string terminator. An external routine can freely change the content of a string. It cannot, however, write beyond the end of the null terminator.
ByRef Variant	A pointer to a 16-byte variant structure. This structure contains a 2-byte type (the same as that returned by the **VarType** function), followed by 6-bytes of slop (for alignment), followed by 8-bytes containing the value.
ByVal Variant	A 16-byte variant structure. This structure contains a 2-byte type (the same as that returned by the **VarType** function), followed by 6-bytes of slop (for alignment), followed by 8-bytes containing the value.
ByVal Object	For data objects, a 4-byte unsigned long integer. This value can only be used by external routines written specifically for BasicScript. For OLE Automation objects, a 32-bit pointer to an **LPDISPATCH** handle is passed.
ByRef Object	For data objects, a pointer to a 4-byte unsigned long integer that references the object. This value can only be used by external routines written specifically for BasicScript. For OLE Automation objects, a pointer an LPDISPATCH value is passed.
ByVal User-defined type	The entire structure is passed to the external routine. It is important to remember that structures in BasicScript are packed on 2-byte boundaries, meaning that the individual structure members may not be aligned consistently with similar structures declared in C.
ByRef User-defined type	A pointer to the structure. It is important to remember that structures in BasicScript are packed on 2-byte boundaries, meaning that the individual structure members may not be aligned consistently with similar structures declared in C.

Data type	Is passed as
Arrays	A pointer to a packed array of elements of the given type. Arrays can only be passed by reference.
Dialogs	Dialogs cannot be passed to external routines.

Only variable-length strings can be passed to external routines; fixed-length strings are automatically converted to variable-length strings.

BasicScript passes data to external functions consistent with that routine's prototype as defined by the **Declare** statement. There is one exception to this rule: you can override **ByRef** parameters using the **ByVal** keyword when passing individual parameters. The following example shows a number of different ways to pass an **Integer** to an external routine called Foo:

```
Declare Sub Foo Lib "MyLib" (ByRef i As Integer)
Sub Main
   Dim i As Integer
   i = 6
   Foo 6          'Passes a temporary integer (value 6) by
                  'reference
   Foo i          'Passes variable "i" by reference
   Foo (i)        'Passes a temporary integer (value 6) by
                  'reference
   Foo i + 1      'Passes temporary integer (value 7) by
                     'reference
   Foo ByVal i       'Passes i by value
End Sub
```

The above example shows that the only way to override passing a value by reference is to use the **ByVal** keyword.

Note: Use caution when using the **ByVal** keyword in this way. The external routine **Foo** expects to receive a pointer to an **Integer**—a 32-bit value; using **ByVal** causes BasicScript to pass the **Integer** by value—a 16-bit value. Passing data of the wrong size to any external routine will have unpredictable results.

Returning Values from External Routines

BasicScript supports the following values returned from external routines: **Integer**, **Long**, **Single**, **Double**, **String**, **Boolean**, and all object types. When returning a **String**, BasicScript assumes that the first null-terminator is the end of the string.

Calling External Routines in Multi-Threaded Environments

In multi-threaded environments (such as Win32), BasicScript makes a copy of all data passed to external routines. This allows other simultaneously executing scripts to continue executing before the external routine returns.

Care must be exercised when passing a the same by-reference variable twice to external routines. When returning from such calls, BasicScript must update the real data from the copies made prior to calling the external function. Since the same variable was passed twice, you will be unable to determine which variable will be updated.

Example

```
Declare Function IsLoaded% Lib "Kernel" _
   Alias "GetModuleHandle" (ByVal name$)
Declare Function GetProfileString Lib "Kernel" _
   (ByVal SName$,ByVal KName$,_
   ByVal Def$,ByVal Ret$,ByVal Size%) As Integer
Sub Main()
   SName$ = "Intl"'Win.ini section name.
   KName$ = "sCountry"'Win.ini country setting.
   ret$ = String$(255, 0)'Initialize return string.
   If GetProfileString(SName$,KName$,"",ret$,Len(ret$)) Then
      MsgBox "Your country setting is: " & ret$
   Else
      MsgBox "There is no country setting in your win.ini file."
   End If
   If IsLoaded("Progman") Then
      MsgBox "Progman is loaded."
   Else
      MsgBox "Progman is not loaded."
   End If
End Sub
```

See Also **Call** (statement); **Sub**...**End Sub** (statement); **Function**...**End Function** (statement).

Platform(s) All platforms support **Declare** for forward referencing.

The following platforms currently support the use of **Declare** for referencing external routines: Windows, Win32/Intel, Win32/PPC, Macintosh, OS/2, NetWare, and some UNIX platforms. See below for details.

Platform Notes **Windows:** Under Windows, external routines are contained in DLLs. The libraries containing the routines are loaded when the routine is called for the first time (i.e., not when the script is loaded). This allows a script to reference external DLLs that potentially do not exist.

All the Windows API routines are contained in DLLs, such as "user", "kernel", and "gdi". The file extension ".exe" is implied if another extension is not given.

If the *LibName$* parameter does not contain an explicit path to the DLL, the following search will be performed for the DLL (in this order):

1. The current directory
2. The Windows directory
3. The Windows system directory
4. The directory containing BasicScript
5. All directories listed in the path environment variable

If the first character of *AliasName$* is #, then the remainder of the characters specify the ordinal number of the routine to be called. For example, the following two statements are equivalent (under Windows, **GetCurrentTime** is defined as ordinal 15 in the user.exe module):

```
Declare Function GetTime Lib "user" Alias "GetCurrentTime" ()
As Integer
Declare Function GetTime Lib "user" Alias "#15" () As Integer
```

Under Windows, the names of external routines declared using the **CDecl** keyword are usually preceded with an underscore character. When BasicScript searches for your external routine by name, it first attempts to load the routine exactly as specified. If unsuccessful, BasicScript makes a second attempt by prepending an underscore character to the specified name. If both attempts fail, then BasicScript generates a runtime error. Under Windows, external routines declared using the **Pascal** keyword are case insensitive, whereas external routines declared using the **CDecl** keyword are case sensitive.

Windows has a limitation that prevents **Double**, **Single**, and **Date** values from being returned from routines declared with the **CDecl** keyword. Routines that return data of these types should be declared **Pascal**.

BasicScript does not perform an increment on OLE automation objects before passing them to external routines.

Platform Notes

Win32: Under Win32, eternal routines are contained in DLLs. The libraries containing the routines are loaded when the routine is called for the first time (i.e., not when the script is loaded). This allows a script to reference external DLLs that potentially do not exist.

Note: You cannot execute routines contained in 16-bit Windows DLLs from the 32-bit version of BasicScript.

All the Win32 API routines are contained in DLLs, such as "user32", "kernel32", and "gdi32". The file extension ".exe" is implied if another extension is not given.

The **Pascal** and **StdCall** calling conventions are identical on Win32 platforms. Furthermore, on this platform, the arguments are passed using C ordering regardless of the calling convention—right to left on the stack.

If the *LibName$* parameter does not contain an explicit path to the DLL, the following search will be performed for the DLL (in this order):

1. The directory containing BasicScript
2. The current directory
3. The Windows system directory
4. The Windows directory
5. All directories listed in the path environment variable

If the first character of *AliasName$* is #, then the remainder of the characters specify the ordinal number of the routine to be called. For example, the following two statements are equivalent (under Win32, **GetCurrentTime** is defined as **GetTickCount**, ordinal 300, in kernel32.dll):

```
Declare Function GetTime Lib "kernel32.dll" Alias
"GetTickCount" () As Long
Declare Function GetTime Lib "kernel32.dll" Alias "#300" () As
Long
```

Under Win32, *name* and *AliasName$* are case-sensitive.

Under Win32, all string passed by value are converted to MBCS strings. Similarly, any string returned from an external routine is assumes to be a null-terminated MBCS string.

BasicScript does not perform an increment on OLE automation objects before passing them to external routines. When returned from an external function, BasicScript assumes that the properties and methods of the OLE automation object are UNICODE and that the object uses the default system locale.

Platform Notes **NetWare:** Under NetWare, external routines are contained within NLMs. If no file extension is specified in *LibName$*, then ".nlm" is assumed.

Since the standard C library is implemented as an NLM under NetWare, it is possible to call many C routines directly from BasicScript. For example, the following code calls **Printf** with a **String** and an **Integer**:

```
Declare Sub Printf Lib "CLIB.NLM" (ByVal F$,ByVal s$,ByVal i%)
Sub Main()
   Printf "Hello, ","world.",10
End Sub
```

If *LibName$* does not contain an explicit path, then NetWare looks in the system directory. The NLM specified by *LibName$* is loaded when the first call to an external in that module is accessed, thus allowing execution of scripts containing calls to NLMs that do not exist. (If the NLM is already loaded, then no work is done.)

Under NetWare, the *name* and *AliasName$* parameters are case-sensitive.

Platform Notes **Macintosh:** On the Macintosh, external routines are contained in code fragments as specified by the **LibName$** parameter. BasicScript uses the following rules for locating your code fragment:

1. If *LibName$* contains an explicit path, that code fragment will be loaded.
2. If no path is specified in *LibName$*, then BasicScript will look in the folder containing BasicScript, then the System folder.
3. If both of the above fail, then BasicScript will search for a code fragment whose CFRG resource name is the same as *LibName$*. The search is performed in the folder containing BasicScript, then the System folder.

The name is compared case-sensitive.

The *name*, *AliasName$*, and *LibName$* parameters are case-sensitive.

For more information on the calling conventions for code fragments, Apple publishes the following books:

1. Inside Macintosh: PowerPC System Software
2. Building CFM-68K Runtime Programs for Macintosh Computers

Platform Notes **OS/2:** If the *LibName$* parameter does not contain an explicit path to the DLL, the following search will be performed for the DLL (in this order):

1. The current directory.
2. All directories listed in the path environment variable.

The **Declare** statement under OS/2 supports calling both 16-bit and 32-bit routines. The following table shows how this relates to the supported calling conventions:

Calling Convention	Supports 16-Bit Calls	Supports 32-Bit Calls
System	No	Yes
Pascal	Yes	Yes
CDec1	Yes	No

Note: BasicScript does not support passing of **Single** and **Double** values to external 16-bit subroutines or functions. These data types are also not supported as return values from external 16-bit functions.

If the first character of *AliasName$* is #, then the remainder of the characters specify the ordinal number of the routine to be called. The following example shows an ordinal used to access the **DosQueryCurrentDisk** function contained in the doscall1.dll module:

```
Declare Function System DosQueryCurrentDisk Lib "doscall1.dll"
Alias "#275" _
(ByRef Drive As Long,ByRef Map As Long) As Integer
```

Under OS/2, the *name* and *AliasName$* parameters are case-sensitive.

Note: All external routines contained in the doscall1.dll module require the use of an ordinal.

Platform Notes

UNIX: The Declare statement can be used to reference routines contained in shared libraries on the following UNIX platforms: HP-UX, Solaris.

If *LibPath$* does not contain an explicit path, then a search is made for the shared library in each path in the colon separated list as specified by the following environment variable:

Platform	Environment Variable
HP-UX	SHLIB_PATH
Solaris	LD_LIBRARY_PATH

The following example shows how to call the printf function on the HP-UX platform:

```
Declare Sub PrintString Lib "/lib/libc.sl" Alias "_printf" _
   (ByVal FormatString As String,ByVal s As String)
Sub Main
   PrintString "Hello, ","world."
End Sub
```

A special note when passing **Single** values to external routines on HP-UX: When passing **Single** values to external routines compiled in ANSI mode, the parameter in the **Declare** statement should be specified as **Double**. External routines compiled in K&R mode should have float parameters defined as **Single** as normal. This is due to calling convention differences between these two standards: In ANSI mode, floats are promoted to double prior to passing.

DefType (statement)

Syntax

```
DefInt letterrange
DefLng letterrange
DefStr letterrange
DefSng letterrange
DefDbl letterrange
DefCur letterrange
DefObj letterrange
DefVar letterrange
DefBool letterrange
DefDate letterrange
```

Description

Establishes the default type assigned to undeclared or untyped variables.

Comments

The **Def***Type* statement controls automatic type declaration of variables. Normally, if a variable is encountered that hasn't yet been declared with the **Dim**, **Public**, or **Private** statement or does not appear with an explicit type-declaration character, then that variable is declared implicitly as a variant (**DefVar** A–Z). This can be changed using the **Def***Type* statement to specify starting letter ranges for *Type* other than integer. The *letterrange* parameter is used to specify starting letters. Thus, any variable that begins with a specified character will be declared using the specified *Type*.

The syntax for *letterrange* is:

letter `[-`*letter*`]` `[,`*letter* `[-`*letter*`]]...`

Def*Type* variable types are superseded by an explicit type declaration—using either a type-declaration character or the **Dim**, **Public**, or **Private** statement.

The **Def***Type* statement only affects how BasicScript compiles scripts and has no effect at runtime.

The **Def***Type* statement can only appear outside all **Sub** and **Function** declarations.

The following table describes the data types referenced by the different variations of the **Def***Type* statement:

Statement	Data Type
DefInt	**Integer**
DefLng	**Long**

Statement	Data Type
DefStr	**String**
DefSng	**Single**
DefDbl	**Double**
DefCur	**Currency**
DefObj	**Object**
DefVar	**Variant**
DefBool	**Boolean**
DefDate	**Date**

Example

```
DefStr a-l
DefLng m-r
DefSng s-u
DefDbl v-w
DefInt x-z
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a = 100.52
   m = 100.52
   s = 100.52
   v = 100.52
   x = 100.52
   message = "The values are:"
   message = message & "(String) a: " & a
   message = message & "(Long) m: " & m
   message = message & "(Single) s: " & s
   message = message & "(Double) v: " & v
   message = message & "(Integer) x: " & x
   MsgBox message
End Sub
```

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Integer** (data type).

Platform(s) All.

DeleteSetting (statement)

Syntax `DeleteSetting` *appname* `[,`*section* `[,`*key*`]]`

Description Deletes a setting from the registry.

Comments

You can control the behavior of **DeleteSetting** by omitting parameters. If you specify all three parameters, then **DeleteSetting** deletes your specified setting. If you omit *key*, then **DeleteSetting** deletes all of the keys from *section*. If both *section* and *key* are omitted, then **DeleteSetting** removes that application's entry from the system registry.

The following table describes the named parameters to the **DeleteSetting** statement:

Named Parameter	Description
appname	**String** expression indicating the name of the application whose setting will be deleted.
section	**String** expression indicating the name of the section whose setting will be deleted.
key	**String** expression indicating the name of the setting to be deleted from the registry.

Example

```
'The following example adds two entries to the Windows registry
'if run under Win32 or to NEWAPP.INI on other platforms,
'using the SaveSetting statement. It then uses DeleteSetting
'first to remove the Startup section, then to remove
'the NewApp key altogether.

Sub Main()
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Height", setting := 200
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Width", setting := 320

   DeleteSetting "NewApp", "Startup"'Remove Startup section
   DeleteSetting "NewApp"           'Remove NewApp key
End Sub
```

See Also

SaveSetting (statement); **GetSetting** (function); **GetAllSettings** (function).

Platform(s)

Windows, Win32, OS/2.

Platform Notes

Win32: Under Win32, this statement operates on the system registry. All settings are saved under the following entry in the system registry:

```
HKEY_CURRENT_USER\Software\BasicScript Program
Settings\appname\section\key
```

Windows, OS/2: Settings are stored in INI files. The name of the INI file is specified by *appname*. If *appname* is omitted, then this command operates on the WIN.INI file. For example, to delete the **sLanguage** setting from the **intl** section of the WIN.INI file, you could use the following statement:

```
s$ = DeleteSetting(,"intl","sLanguage")
```

Desktop.ArrangeIcons (method)

Syntax `Desktop.ArrangeIcons`

Description Reorganizes the minimized applications on the desktop.

Example
```
Sub Main()
   Desktop.ArrangeIcons
End Sub
```

See Also **Desktop.Cascade** (method); **Desktop.Tile** (method).

Platform(s) Windows.

Desktop.Cascade (method)

Syntax `Desktop.Cascade`

Description Cascades all non-minimized windows.

Example
```
'This example cascades all the windows on the desktop. It first
'restores any minimized applications so that they are included
'in the cascade.
Sub Main()
   Dim apps$()
   AppList apps$
   For i = LBound(apps) To UBound(apps)
      AppRestore apps(i)
   Next i
   Desktop.Cascade
End Sub
```

See Also **Desktop.Tile** (method); **Desktop.ArrangeIcons** (method).

Platform(s) Windows.

Desktop.SetColors (method)

Syntax `Desktop.SetColors` *ControlPanelItemName$*

Description Changes the system colors to one of a predefined color set.

Example
```
'This example allows the user to select any of the available
'Windows color schemes.
Sub Main()
   'Get color schemes from Windows
   Dim names$()
```

```
   ReadINISection "color schemes",names$,"CONTROL.INI"
SelectAgain:'Allow user to select color scheme
   item = SelectBox("Set Colors","Available Color Sets:",names$)
   If item <> -1 Then
      Desktop.SetColors names$(item)
      Goto SelectAgain
   End If
End Sub
```

See Also **Desktop.SetWallpaper** (method).

Platform(s) Windows.

Platform Notes **Windows:** Under Windows, the names of the color sets are contained in the control.ini file.

Desktop.SetWallpaper (method)

Syntax `Desktop.SetWallpaper` *filename$, isTile*

Description Changes the desktop wallpaper to the bitmap specified by *filename$*.

Comments The wallpaper will be tiled if *isTile* is True; otherwise, the bitmap will be centered on the desktop.

To remove the wallpaper, set the *filename$* parameter to "", as in the following example:

```
Desktop.SetWallpaper "",True
```

Example

```
'This example reads a list of .BMP files from the Windows
'directory and allows the user to select any of these as
'wallpaper.
Sub Main()
   Dim list$()
   ' Create the prefix for the bitmap filenames
   d$ = System.WindowsDirectory$
   If Right(d$,1) <> "\" Then d$ = d$ & "\"
   f$ = d$ & "*.BMP"
   FileList list$,f$'Get list of bitmaps from Windows directory
   'Were there any bitmaps?
   If ArrayDims(list$) = 0 Then
      MsgBox "There aren't any bitmaps in the Windows directory"
      Exit Sub
   End If
   'Add "(none)".
   ReDim Preserve list$ (UBound(list$) + 1)
   list$(UBound(list$)) = "(none)"
SelectAgain:'Allow user to select item
```

```
   item = SelectBox("Set Wallpaper","Available Wallpaper:",list$)
   Select Case item
      Case -1
         End
      Case UBound(list$)
         Desktop.SetWallPaper "",True
         Goto SelectAgain
      Case Else
         Desktop.SetWallPaper d$ & list$(item),True
         Goto SelectAgain
   End Select
End Sub
```

See Also **Desktop.SetColors** (method).

Platform(s) Windows.

Platform Notes **Windows:** Under Windows, the **Desktop.SetWallpaper** method makes permanent changes to the wallpaper by writing the new wallpaper information to the win.ini file.

Desktop.Snapshot (method)

Syntax `Desktop.Snapshot [`*spec*`]`

Description Takes a snapshot of a particular section of the screen and saves it to the Clipboard.

Comments The *spec* parameter is an **Integer** specifying the screen area to be saved. It can be any of the following:

0	Entire screen
1	Client area of the active application
2	Entire window of the active application
3	Client area of the active window
4	Entire window of the active window

Before the snapshot is taken, each application is updated. This ensures that any application that is in the middle of drawing will have a chance to finish before the snapshot is taken.

There is a slight delay if the specified window is large.

Example

```
'This example takes a snapshot of Program Manager and pastes
'the resulting bitmap into Windows Paintbrush.
Sub Main()
   AppActivate "Program Manager"'Activate Program Manager.
```

```
   Desktop.Snapshot 2'Place snapshot into Clipboard.
   id = Shell("pbrush")'Run Paintbrush.
   Menu "Edit.Paste"'Paste snapshot into Paintbrush.
End Sub
```

Platform(s) Windows.

Platform Notes **Windows:** Under Windows, pictures are placed into the Clipboard in bitmap format.

Desktop.Tile (method)

Syntax `Desktop.Tile`

Description Tiles all non-minimized windows.

Example

```
'This example tiles all the windows on the desktop. It first
'restores any minimized applications so that they are included
'in the tile.
Sub Main()
   Dim apps$()
   AppList apps$
   For i = LBound(apps) To UBound(apps)
      AppRestore apps(i)
   Next i
   Desktop.Tile
End Sub
```

See Also **Desktop.Cascade** (method); **Desktop.ArrangeIcons** (method).

Platform(s) Windows.

Dialog (function)

Syntax `Dialog(`*DialogVariable* `[,[`*DefaultButton*`] [,`*Timeout*`]])`

Description Displays the dialog box associated with *DialogVariable*, returning an **Integer** indicating which button was clicked.

Comments The **Dialog** function returns any of the following values:

–1	The OK button was clicked.
0	The Cancel button was clicked.

>0	A push button was clicked. The returned number represents which button was clicked based on its order in the dialog box template (1 is the first push button, 2 is the second push button, and so on).

The **Dialog** function accepts the following parameters:

Parameter	Description
DialogVariable	Name of a variable that has previously been dimensioned as a user dialog box. This is accomplished using the **Dim** statement: `Dim MyDialog As MyTemplate` All dialog variables are local to the **Sub** or **Function** in which they are defined. Private and public dialog variables are not allowed.
DefaultButton	An **Integer** specifying which button is to act as the default button in the dialog box. The value of *DefaultButton* can be any of the following –1 — This value indicates that the OK button, if present, should be used as the default. 0 — This value indicates that the Cancel button, if present, should be used as the default. >0 — This value indicates that the *N*th button should be used as the default. This number is the index of a push button within the dialog box template. If *DefaultButton* is not specified, then –1 is used. If the number specified by *DefaultButton* does not correspond to an existing button, then there will be no default button. The default button appears with a thick border and is selected when the user presses Enter on a control other than a push button.
Timeout	An **Integer** specifying the number of milliseconds to display the dialog box before automatically dismissing it. If *Timeout* is not specified or is equal to 0, then the dialog box will be displayed until dismissed by the user. If a dialog box has been dismissed due to a timeout, the **Dialog** function returns 0.

A runtime error is generated if the dialog template specified by **DialogVariable** does not contain at least one of the following statements:

```
   PushButton        CancelButton
   OKButton          PictureButton
```

Example

```
'This example displays an abort/retry/ignore disk error dialog
'box.
Sub Main()
   Begin Dialog DiskErrorTemplate 16,32,152,48,"Disk Error"
      Text 8,8,100,8,"The disk drive door is open."
      PushButton 8,24,40,14,"Abort",.Abort
      PushButton 56,24,40,14,"Retry",.Retry
      PushButton 104,24,40,14,"Ignore",.Ignore
   End Dialog
   Dim DiskError As DiskErrorTemplate
   r% = Dialog(DiskError,3,0)
   MsgBox "You selected button: " & r%
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Dialog (statement)

Syntax `Dialog` *DialogVariable* `[,[`*DefaultButton*`] [,`*Timeout*`]]`

Description Same as the **Dialog** function, except that the **Dialog** statement does not return a value. (See **Dialog** [function].)

Example

```
'This example displays an abort/retry/ignore disk error dialog
'box.
Sub Main()
   Begin Dialog DiskErrorTemplate 16,32,152,48,"Disk Error"
      Text 8,8,100,8,"The disk drive door is open."
      PushButton 8,24,40,14,"Abort",.Abort
      PushButton 56,24,40,14,"Retry",.Retry
      PushButton 104,24,40,14,"Ignore",.Ignore
   End Dialog
   Dim DiskError As DiskErrorTemplate
   Dialog DiskError,3,0
End Sub
```

See Also **Dialog** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Dialogs (topic)

Dialogs are supported on the following platforms: Windows, Win32, OS/2, UNIX, and Macintosh. The following table describes the default font use by BasicScript to display all runtime dialogs:

Default Font in Dialog Boxes

Platform	Default Font
Windows	For non-MBCS systems, BasicScript uses the 8-point MS Sans Serif font. For MBCS systems, BasicScript uses the default system font.
Win32	For non-MBCS systems, BasicScript uses the 8-point MS Sans Serif font. For MBCS systems, BasicScript uses the default system font.
Macintosh	10-point Geneva.
UNIX	The default font is determined by X resource files (e.g., **$HOME/.xdefaults**).

When Help is enabled within a dialog, the help key is enabled as described in the following table:

Help Key in BasicScript Dialogs

Platform	Help Key
Windows	F1
Win32	F1
OS/2	F1
Macintosh	Command+?
UNIX	The default help key is F1, unless if has been redefined in your X resource files.

Dim (statement)

Syntax `Dim` *name* `[(`<*subscripts*>`)] [As [New]` *type*`] [,`*name* `[(`<*subscripts*>`)] [As [New]` *type*`]]...`

Description Declares a list of local variables and their corresponding types and sizes.

Comments

If a type-declaration character is used when specifying *name* (such as %, @, &, $, or !), the optional [**As** *type*] expression is not allowed. For example, the following are allowed:

```
Dim Temperature As Integer
Dim Temperature%
```

The *subscripts* parameter allows the declaration of dynamic and fixed arrays. The *subscripts* parameter uses the following syntax:

```
[lower to] upper [,[lower to] upper]...
```

The *lower* and *upper* parameters are integers specifying the lower and upper bounds of the array. If *lower* is not specified, then the lower bound as specified by **Option Base** is used (or 1 if no **Option Base** statement has been encountered). BasicScript supports a maximum of 60 array dimensions.

The total size of an array (not counting space for strings) is limited to 64K.

Dynamic arrays are declared by not specifying any bounds:

```
Dim a()
```

The *type* parameter specifies the type of the data item being declared. It can be any of the following data types: **String**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Object**, data object, built-in data type, or any user-defined data type. When specifying explicit object types, you can use the following syntax for *type*:

```
module.class
```

Where *module* is the name of the module in which the object is defined and *class* is the type of object. For example, to specify the OLE automation variable for Excel's Application object, you could use the following code:

```
Dim a As Excel.Application
```

Note: Explicit object types can only be specified for data objects and early bound OLE automation objects—i.e., objects whose type libraries have been registered with BasicScript.

A **Dim** statement within a subroutine or function declares variables local to that subroutine or function. If the **Dim** statement appears outside of any subroutine or function declaration, then that variable has the same scope as variables declared with the **Private** statement.

Fixed-Length Strings

Fixed-length strings are declared by adding a length to the **String** type-declaration character:

```
Dim name As String * length
```

where *length* is a literal number specifying the string's length.

Implicit Variable Declaration

If BasicScript encounters a variable that has not been explicitly declared with **Dim**, then the variable will be implicitly declared using the specified type-declaration character (#, %, @, $, or &). If the variable appears without a type-declaration character, then the first letter is matched against any pending **Def***Type* statements, using the specified type if found. If no **Def***Type* statement has been encountered corresponding to the first letter of the variable name, then **Variant** is used.

Declaring Explicit OLE Automation Objects

The Dim statement can be used to declare variables of an explicit object type for objects known to BasicScript through type libraries. This is accomplished using the following syntax:

```
Dim name As application.class
```

The *application* parameter specifies the application used to register the OLE automation object and *class* specifies the specific object type as defined in the type library. Objects declared in this manner are early bound, meaning that the BasicScript is able resolve method and property information at compile time, improving the performance when invoking methods and properties off that object variable.

Creating New Objects

The optional **New** keyword is used to declare a new instance of the specified data object. This keyword cannot be used when declaring arrays or OLE automation objects.

At runtime, the application or extension that defines that object type is notified that a new object is being defined. The application responds by creating a new physical object (within the appropriate context) and returning a reference to that object, which is immediately assigned to the variable being declared.

When that variable goes out of scope (i.e., the **Sub** or **Function** procedure in which the variable is declared ends), the application is notified. The application then performs some appropriate action, such as destroying the physical object.

Initial Values

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0
Long	0
Double	0.0
Single	0.0
Date	December 31, 1899 00:00:00
Currency	0.0

Data Type	Initial Value
Boolean	**False**
Object	**Nothing**
Variant	**Empty**
String	"" (zero-length string)
User-defined type	Each element of the structure is given an initial value, as described above.
Arrays	Each element of the array is given an initial value, as described above.

Naming Conventions

Variable names must follow these naming rules:

1. Must start with a letter.
2. May contain letters, digits, and the underscore character (_); punctuation is not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character.
3. The last character of the name can be any of the following type-declaration characters: #, @, %, !, &, and $.
4. Must not exceed 80 characters in length.
5. Cannot be a reserved word.

Examples

```
'The following examples use the Dim statement to declare various
'variable types.
Sub Main()
      Dim i As Integer
      Dim l&                    'Long
      Dim s As Single
      Dim d#                    'Double
      Dim c$                    'String
      Dim MyArray(10) As Integer'10 element integer array
      Dim MyStrings$(2,10)'2-10 element string arrays
      Dim Filenames$(5 to 10)'6 element string array
      Dim Values(1 to 10, 100 to 200)'111 element variant array
End Sub
```

See Also **Redim** (statement); **Public** (statement); **Private** (statement); **Option Base** (statement).

Platform(s) All.

Dir, Dir$ (functions)

Syntax

```
Dir[$] [(pathname [,attributes])]
Dir[$] [(pathname, filetype [,attributes])]
```

Description

Returns a **String** containing the first or next file matching *pathname*.

If *pathname* is specified, then the first file matching that *pathname* is returned. If *pathname* is not specified, then the next file matching the initial *pathname* is returned.

Comments

Dir$ returns a **String**, whereas **Dir** returns a **String** variant.

The **Dir$/Dir** functions take the following named parameters:

Named Parameter	Description
pathname	**String** containing a file specification. If this parameter is specified, then **Dir$** returns the first file matching this file specification. If this parameter is omitted, then the next file matching the initial file specification is returned. If no path is specified in *pathname*, then all files are returned from the current directory. An error is generated if *pathname$* is **Null**.
filetype	Indicates the type of file to return. If *pathname* is also specified, then files of this type are returned from that directory. Otherwise, files of this type are returned from the current directory. File types are specified using the MacID function.
attributes	**Integer** specifying attributes of files you want included in the list, as described below. If this parameter is omitted, then only the normal, read-only, and archive files are returned.

An error is generated if **Dir$** is called without first calling it with a valid *pathname*.

If there is no matching *pathname*, then a zero-length string is returned.

Wildcards

The *pathname* argument can include wildcards, such as * and ?. The * character matches any sequence of zero or more characters, whereas the ? character matches any single character. Multiple *'s and ?'s can appear within the expression to form complete searching patterns. The following table shows some examples:

This pattern	Matches these files	Doesn't match these files
S.TXT	SAMPLE.TXT	SAMPLE
	GOOSE.TXT	SAMPLE.DAT
	SAMS.TXT	
C*T.TXT	CAT.TXT	CAP.TXT
		ACATS.TXT
C*T	CAT	CAT.DOC
	CAP.TXT	
C?T	CAT	CAT.TXT
	CUT	CAPIT
		CT
*	(All files)	

Attributes

You can control which files are included in the search by specifying the optional attributes parameter. The **Dir**, **Dir$** functions always return all normal, read-only, and archive files (**ebNormal Or ebReadOnly Or ebArchive**). To include additional files, you can specify any combination of the following attributes (combined with the **Or** operator):

Constant	Value	Includes
ebNormal	0	Read-only, archive, subdir, and none
ebHidden	2	Hidden files
ebSystem	4	System files
ebVolume	8	Volume label
ebDirectory	16	Subdirectories

Example

```
'This example dimensions a null array and fills it with
'directory entries. The result is displayed in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim a$(10)
   a(1) = Dir$("*.*")
```

```
    i% = 1
    While (a(i%) <> "") And (i% < 10)
       i% = i% + 1
       a(i%) = Dir$
    Wend
    MsgBox a(1) & crlf & a(2) & crlf & a(3) & crlf & a(4)
End Sub
```

See Also **ChDir** (statement); **ChDrive** (statement); **CurDir**, **CurDir$** (functions); **MkDir** (statement); **RmDir** (statement); **FileList** (statement).

Platform(s) All.

Platform Notes **Macintosh:** The Macintosh does not support wildcard characters such as * and ?. These are valid filename characters. Instead of wildcards, the Macintosh uses the **MacID** function to specify a collection of files of the same type. The syntax for this function is:

Dir$(*pathname*,MacID(*text$*) [,*attributes*])

The *text$* parameter is a four-character string containing a file type, a resource type, an application signature, or an Apple event. A runtime error occurs if the **MacID** function is used on platforms other than the Macintosh.

When the **MacID** function is used, the *pathname* parameter specifies the directory in which to search for files of the indicated type.

Platform Notes **Windows:** For compatibility with DOS wildcard matching, BasicScript special-cases the pattern "*.*" to indicate all files, not just files with a periods in their names.

UNIX: On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

DiskDrives (statement)

Syntax DiskDrives *array*()

Description Fills the specified **String** or **Variant** array with a list of valid drive letters.

Comments The *array*() parameter specifies either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed.

If *array*() is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions.

If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for **String** arrays) or **Empty** (for **Variant** arrays). A runtime error results if the array is too small to hold the new elements.

Example

```
'This example builds and displays an array containing the first
'three available disk drives.
Sub Main()
   Dim drive$()
   DiskDrives drive$
   r% = SelectBox("Available Disk Drives",,drive$)
End Sub
```

See Also **ChDrive** (statement); **DiskFree** (function).

Platform(s) Windows, Win32, NetWare.

Platform Notes **NetWare:** Under NetWare, this command returns a list of volume names.

DiskFree (function)

Syntax `DiskFree&([drive$])`

Description Returns a **Long** containing the free space (in bytes) available on the specified drive.

Comments If *drive$* is zero-length or not specified, then the current drive is assumed.

Only the first character of the *drive$* string is used.

On systems that do not support drive letters, the *drive$* parameter specifies the name of the path from which to retrieve the free disk space.

Example

```
'This example uses DiskFree to set the value of i and then
'displays the result in a message box.
Sub Main()
   s$ = "c"
   i# = DiskFree(s$)
   MsgBox "Free disk space on drive '" & s$ & "' is: " & i#
End Sub
```

See Also **ChDrive** (statement); **DiskDrives** (statement).

Platform(s) All.

Platform Notes **NetWare:** Since NetWare does not support drive letters, the *drive$* parameter specifies a volume name (up to 14 characters).

DlgCaption (function)

Syntax `DlgCaption[()]`

Description Returns a string containing the caption of the active user-defined dialog box.

Comments This function returns a zero-length string if the active dialog has no caption.

See Also **Begin Dialog** (statement).

Platform(s) Windows, Win32, Macintosh, UNIX, OS/2.

DlgCaption (statement)

Syntax `DlgCaption` *text*

Description Changes the caption of the current dialog to *text*.

Example

```
'This example displays a dialog box, adjusting the caption to
'contain the text of the currently selected option button.
Function DlgProc(c As String,a As Integer,v As Integer)
   If a = 1 Then
      DlgCaption choose(DlgValue("OptionGroup1") + 1, _
         "Blue","Green")
   ElseIf a = 2 Then
      DlgCaption choose(DlgValue("OptionGroup1") + 1, _
         "Blue","Green")
   End If
End Function
Sub Main()
   Begin Dialog UserDialog ,,149,45,"Untitled",.DlgProc
      OKButton 96,8,40,14
      OptionGroup .OptionGroup1
         OptionButton 12,12,56,8,"Blue",.OptionButton1
         OptionButton 12,28,56,8,"Green",.OptionButton2
   End Dialog
   Dim d As UserDialog
   Dialog d
End Sub
```

See Also **Begin Dialog** (statement).

Platform(s) Windows, Win32, Macintosh, UNIX, OS/2.

DlgControlId (function)

Syntax `DlgControlId(`*ControlName$*`)`

Description Returns an **Integer** containing the index of the specified control as it appears in the dialog box template.

Comments The first control in the dialog box template is at index 0, the second is at index 1, and so on.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with that control in the dialog box template.

The BasicScript statements and functions that dynamically manipulate dialog box controls identify individual controls using either the *.Identifier* name of the control or the control's index. Using the index to refer to a control is slightly faster but results in code that is more difficult to maintain.

Example

```
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
   'If a control is clicked, disable the next three controls.
   If Action% = 2 Then
      'Enable the next three controls.
      start% = DlgControlId(ControlName$)
      For i = start% + 1 To start% + 3
         DlgEnable i,True
      Next i
      DlgProc = 1'Don't close the dialog box.
   End If
End Function
```

See Also **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgEnable (function)

Syntax `DlgEnable(`*ControlName$* `|` *ControlIndex*`)`

Description Returns **True** if the specified control is enabled; returns **False** otherwise.

Comments Disabled controls are dimmed and cannot receive keyboard or mouse input.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

If you attempt to disable the control with the focus, BasicScript will automatically set the focus to the next control in the tab order.

Example

```
If DlgEnable("SaveOptions") Then
```

```
   MsgBox "The Save Options are enabled."
End If
If DlgEnable(10) And DlgVisible(12) Then code = 1 Else code = 2
```

See Also **DlgControlId** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgEnable (statement)

Syntax `DlgEnable {`*ControlName$* `|` *ControlIndex*`} [,`*isOn*`]`

Description Enables or disables the specified control.

Comments Disabled controls are dimmed and cannot receive keyboard or mouse input.

The *isOn* parameter is an **Integer** specifying the new state of the control. It can be any of the following values:

0	The control is disabled.
1	The control is enabled.
Omitted	Toggles the control between enabled and disabled.

Option buttons can be manipulated individually (by specifying an individual option button) or as a group (by specifying the name of the option group).

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example

```
DlgEnable "SaveOptions", False'Disable the Save Options control.
DlgEnable "EditingOptions"'Toggle a group of option buttons.
For i = 0 To 5
   DlgEnable i,True'Enable six controls.
Next i
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgFocus (function)

Syntax `DlgFocus$[()]`

Description Returns a **String** containing the name of the control with the focus.

Comments The name of the control is the *.Identifier* parameter associated with the control in the dialog box template.

Example

```
'This code fragment makes sure that the control being disabled
'does not currently have the focus (otherwise, a runtime error
'would occur).
If DlgFocus$ = "Files" Then'Does it have the focus?
   DlgFocus "OK"     'Change the focus to another control.
End If
DlgEnable "Files", False'Now we can disable the control.
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgFocus (statement)

Syntax `DlgFocus` *ControlName$* | *ControlIndex*

Description Sets focus to the specified control.

Comments A runtime error results if the specified control is hidden, disabled, or nonexistent.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example

```
'This code fragment makes sure that the control being disabled
'does not currently have the focus (otherwise, a runtime error
'would occur).
If DlgFocus$ = "Files" Then'Does it have the focus?
   DlgFocus "OK"'Change the focus to another control.
End If
DlgEnable "Files", False'Now we can disable the control.
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgListBoxArray (function)

Syntax `DlgListBoxArray({`*ControlName$* `|` *ControlIndex*`}, `*ArrayVariable*`)`

Description Fills a list box, combo box, or drop list box with the elements of an array, returning an **Integer** containing the number of elements that were actually set into the control.

Comments The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

The *ArrayVariable* parameter specifies a single-dimensioned array used to initialize the elements of the control. If this array has no dimensions, then the control will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. *ArrayVariable* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.

Example

```
'This dialog function refills an array with files.
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
   If Action% = 2 And ControlName$ = "Files" Then
      Dim NewFiles$()   'Create a new dynamic array.
      FileList NewFiles$,"*.txt"'Fill the array with files.
      r% = DlgListBoxArray "Files",NewFiles$'Set list box items.
      DlgValue "Files",0'Set the selection to the first item.
      DlgProc = 1    'Don't close the dialog box.
   End If
   MsgBox r% & " items were added to the list box."
End Function
```

See Also **DlgControl**Id (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgListBoxArray (statement)

Syntax `DlgListBoxArray {`*ControlName$* `|` *ControlIndex*`}, `*ArrayVariable*

Description Fills a list box, combo box, or drop list box with the elements of an array.

Comments The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

The *ArrayVariable* parameter specifies a single-dimensioned array used to initialize the elements of the control. If this array has no dimensions, then the control will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. *ArrayVariable* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.

Example

```
'This dialog function refills an array with files.
Function DlgProc(ControlName$,Action%,SuppValue%) As Integer
   If Action% = 2 And ControlName$ = "Files" Then
      Dim NewFiles$()      'Create a new dynamic array.
      FileList NewFiles$,"*.txt"'Fill the array with files.
      DlgListBoxArray "Files",NewFiles$'Set list box items.
```

```
      DlgValue "Files",0'Set the selection to the first item.
   End If
End Function
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgProc (function)

Syntax `Function` *DlgProc*`(`*ControlName$*`,` *Action*`,` *SuppValue*`) As Integer`

Description Describes the syntax, parameters, and return value for dialog functions.

Comments Dialog functions are called by BasicScript during the processing of a custom dialog box. The name of a dialog function (*DlgProc*) appears in the **Begin Dialog** statement as the .*DlgProc* parameter.

Dialog functions require the following parameters:

Parameter	Description
ControlName$	**String** containing the name of the control associated with *Action*.
Action	**Integer** containing the action that called the dialog function.
SuppValue	**Integer** of extra information associated with *Action*. For some actions, this parameter is not used.

When BasicScript displays a custom dialog box, the user may click on buttons, type text into edit fields, select items from lists, and perform other actions. When these actions occur, BasicScript calls the dialog function, passing it the action, the name of the control on which the action occurred, and any other relevant information associated with the action.

The following table describes the different actions sent to dialog functions:

Action	Description
1	This action is sent immediately before the dialog box is shown for the first time. This gives the dialog function a chance to prepare the dialog box for use. When this action is sent, *ControlName$* contains a zero-length string, and *SuppValue* is 0. The return value from the dialog function is ignored in this case.

Action	Description
	Before Showing the Dialog Box After action 1 is sent, BasicScript performs additional processing before the dialog box is shown. Specifically, it cycles though the dialog box controls checking for visible picture or picture button controls. For each visible picture or picture button control, BasicScript attempts to load the associated picture. In addition to checking picture or picture button controls, BasicScript will automatically hide any control outside the confines of the visible portion of the dialog box. This prevents the user from tabbing to controls that cannot be seen. However, it does not prevent you from showing these controls with the **DlgVisible** statement in the dialog function.
2	This action is sent when: • A button is clicked, such as OK, Cancel, or a push button. In this case, *ControlName$* contains the name of the button. *SuppValue* contains 1 if an OK button was clicked and 2 if a Cancel button was clicked; *SuppValue* is undefined otherwise. If the dialog function returns 0 in response to this action, then the dialog box will be closed. Any other value causes BasicScript to continue dialog processing. • A check box's state has been modified. In this case, *ControlName$* contains the name of the check box, and *SuppValue* contains the new state of the check box (1 if on, 0 if off). • An option button is selected. In this case, *ControlName$* contains the name of the option button that was clicked, and *SuppValue* contains the index of the option button within the option button group (0-based). • The current selection is changed in a list box, drop list box, or combo box. In this case, *ControlName$* contains the name of the list box, combo box, or drop list box, and *SuppValue* contains the index of the new item (0 is the first item, 1 is the second, and so on).

Action	Description
3	This action is sent when the content of a text box or combo box has been changed. This action is only sent when the control loses focus. When this action is sent, *ControlName$* contains the name of the text box or combo box, and *SuppValue* contains the length of the new content. The dialog function's return value is ignored with this action.
4	This action is sent when a control gains the focus. When this action is sent, *ControlName$* contains the name of the control gaining the focus, and *SuppValue* contains the index of the control that lost the focus (0-based). The dialog function's return value is ignored with this action.
5	This action is sent continuously when the dialog box is idle. If the dialog function returns 1 in response to this action, then the idle action will continue to be sent. If the dialog function returns 0, then BasicScript will not send any additional idle actions. When the idle action is sent, *ControlName$* contains a zero-length string, and *SuppValue* contains the number of times the idle action has been sent so far.
6	This action is sent when the dialog box is moved. The *ControlName$* parameter contains a zero-length string, and *SuppValue* is 0. The dialog function's return value is ignored with this action.

User-defined dialog boxes cannot be nested. In other words, the dialog function of one dialog box cannot create another user-defined dialog box. You can, however, invoke any built-in dialog box, such as **MsgBox** or **InputBox$**.

Within dialog functions, you can use the following additional BasicScript statements and functions. These statements allow you to manipulate the dialog box controls dynamically.

DlgVisible	**DlgText$**	**DlgText**
DlgSetPicture	**DlgListBoxArray**	**DlgFocus**
DlgEnable	**DlgControlId**	

For compatibility with previous versions of BasicScript, the dialog function can optionally be declared to return a **Variant**. When returning a variable, BasicScript will attempt to convert the variant to an **Integer**. If the returned variant cannot be converted to an **Integer**, then 0 is assumed to be returned from the dialog function.

Example

```
'This dialog function enables/disables a group of option buttons
```

```
'when a check box is clicked.
Function SampleDlgProc(ControlName$, Action%, SuppValue%)
   If Action% = 2 And ControlName$ = "Printing" Then
      DlgEnable "PrintOptions",SuppValue%
      SampleDlgProc = 1'Don't close the dialog box.
   End If
End Function
Sub Main()
   Begin Dialog SampleDialogTemplate _
      34,39,106,45,"Sample",.SampleDlgProc
      OKButton 4,4,40,14
      CancelButton 4,24,40,14
      CheckBox 56,8,38,8,"Printing",.Printing
      OptionGroup .PrintOptions
         OptionButton 56,20,51,8,"Landscape",.Landscape
         OptionButton 56,32,40,8,"Portrait",.Portrait
   End Dialog
   Dim SampleDialog As SampleDialogTemplate
   SampleDialog.Printing = 1
   r% = Dialog(SampleDialog)
End Sub
```

See Also **Begin Dialog** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgSetPicture (statement)

Syntax `DlgSetPicture {`*ControlName$* `|` *ControlIndex*`},`*PictureName$*`,`*PictureType*

Description Changes the content of the specified picture or picture button control.

Comments The **DlgSetPicture** statement accepts the following parameters:

Parameter	Description
ControlName$	**String** containing the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specified control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on). **Note:** When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Parameter	Description
PictureName$	**String** containing the name of the picture. If *PictureType* is 0, then this parameter specifies the name of the file containing the image. If *PictureType* is 10, then *PictureName$* specifies the name of the image within the resource of the picture library. If *PictureName$* is empty, then the current picture associated with the specified control will be deleted. Thus, a technique for conserving memory and resources would involve setting the picture to empty before hiding a picture control.
PictureType	**Integer** specifying the source for the image. The following sources are supported: 0 — The image is contained in a file on disk. 10 — The image is contained in the picture library specified by the **Begin Dialog** statement. When this type is used, the *PictureName$* parameter must be specified with the **Begin Dialog** statement.

Examples

```
'Set picture from a file
DlgSetPicture "Picture1","\windows\checks.bmp",0
'Set control 10's image from a library
DlgSetPicture 27,"FaxReport",10
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function); **Picture** (statement), **PictureButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32:** Under Windows and Win32, picture controls can contain either bitmaps or WMFs (Windows metafiles). When extracting images from a picture library, BasicScript assumes that the resource type for metafiles is 256.

Picture libraries are implemented as DLLs on the Windows and Win32 platforms.

OS/2: Under OS/2, picture controls can contain either bitmaps or Windows metafiles.

Picture libraries under OS/2 are implemented as resources within DLLs. The *PictureName$* parameter corresponds to the name of one of these resources as it appears within the DLL.

Macintosh: Picture controls on the Macintosh can contain only PICT images. These are contained in files of type PICT.

Picture libraries on the Macintosh are files with collections of named PICT resources. The *PictureName$* parameter corresponds to the name of one the resources as it appears within the file..

DlgText (statement)

Syntax `DlgText {`*ControlName$* `|` *ControlIndex*`}, `*NewText$*

Description Changes the text content of the specified control.

Comments The effect of this statement depends on the type of the specified control:

Control Type	Effect of DlgText
Picture	Runtime error.
Option group	Runtime error.
Drop list box	If an exact match cannot be found, the **DlgText** statement searches from the first item looking for an item that starts with *NewText$*. If no match is found, then the selection is removed.
OK button	Sets the label of the control to *NewText$*.
Cancel button	Sets the label of the control to *NewText$*.
Push button	Sets the label of the control to *NewText$*.
List box	Sets the current selection to the item matching *NewText$*. If an exact match cannot be found, the **DlgText** statement searches from the first item looking for an item that starts with *NewText$*. If no match is found, then the selection is removed.
Combo box	Sets the content of the edit field of the combo box to *NewText$*.
Text	Sets the label of the control to *NewText$*.
Text box	Sets the content of the text box to *NewText$*.
Group box	Sets the label of the control to *NewText$*.
Option button	Sets the label of the control to *NewText$*.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example

```
DlgText "GroupBox1","Save Options"  'Change text of group box 1.
If DlgText$(9) = "Save Options" Then
   DlgText 9,"Editing Options"'Change text to "Editing Options".
End If
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgText$ (function)

Syntax `DlgText$(`*ControlName$* `|` *ControlIndex*`)`

Description Returns the text content of the specified control.

Comments The text returned depends on the type of the specified control:

Control Type	Value Returned by DlgText$
Picture	No value is returned. A runtime error occurs.
Option group	No value is returned. A runtime error occurs.
Drop list box	Returns the currently selected item. A zero-length string is returned if no item is currently selected.
OK button	Returns the label of the control.
Cancel button	Returns the label of the control.
Push button	Returns the label of the control.
List box	Returns the currently selected item. A zero-length string is returned if no item is currently selected.
Combo box	Returns the content of the edit field portion of the combo box.
Text	Returns the label of the control.
Text box	Returns the content of the control.
Group box	Returns the label of the control.
Option button	Returns the label of the control.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example

```
MsgBox DlgText$(10)  'Display the text in the tenth control.
If DlgText$("SaveOptions") = "EditingOptions" Then
   MsgBox "You are currently viewing the editing options."
End If
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgValue (function)

Syntax `DlgValue(`*ControlName$* `|` *ControlIndex*`)`

Description Returns an **Integer** indicating the value of the specified control.

Comments The value of any given control depends on its type, according to the following table:

Control Type	D1gValue Returns
Option group	The index of the selected option button within the group (0 is the first option button, 1 is the second, and so on).
List box	The index of the selected item.
Drop list box	The index of the selected item.
Check box	1 if the check box is checked; 0 otherwise.

A runtime error is generated if **DlgValue** is used with controls other than those listed in the above table.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example See **DlgValue** (statement).

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (statement); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgValue (statement)

Syntax `DlgValue {`*ControlName$* `|` *ControlIndex*`},`*Value*

Description Changes the value of the given control.

Comments The value of any given control is an **Integer** and depends on its type, according to the following table:

Control Type	Description of *Value*
Option group	The index of the new selected option button within the group (0 is the first option button, 1 is the second, and so on).
List box	The index of the new selected item.
Drop list box	The index of the new selected item.
Check box	1 if the check box is to be checked; 0 to remove the check.

A runtime error is generated if **DlgValue** is used with controls other than those listed in the above table.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

Example

```
'This code fragment toggles the value of a check box.
If DlgValue("MyCheckBox") = 1 Then
   DlgValue "MyCheckBox",0
Else
   DlgValue "MyCheckBox",1
End If
```

See Also DlgControlId (function); DlgEnable (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText** (function); **DlgValue** (function); **DlgVisible** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgVisible (function)

Syntax `DlgVisible(`*ControlName$* `|` *ControlIndex*`)`

Description Returns **True** if the specified control is visible; returns **False** otherwise.

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

A runtime error is generated if **DlgVisible** is called when no user dialog is active.

Example

```
If DlgVisible("Portrait") Then Beep
If DlgVisible(10) And DlgVisible(12) Then
   MsgBox "The 10th and 12th controls are visible."
End If
```

See Also **DlgControlId** (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText$** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

DlgVisible (statement)

Syntax `DlgVisible {`*ControlName$* `|` *ControlIndex*`} [,`*isOn*`]`

Description Hides or shows the specified control.

Comments Hidden controls cannot be seen in the dialog box and cannot receive the focus using Tab.

The *isOn* parameter is an **Integer** specifying the new state of the control. It can be any of the following values:

1	The control is shown.
0	The control is hidden.
Omitted	Toggles the visibility of the control.

Option buttons can be manipulated individually (by specifying an individual option button) or as a group (by specifying the name of the option group).

The *ControlName$* parameter contains the name of the *.Identifier* parameter associated with a control in the dialog box template. A case-insensitive comparison is used to locate the specific control within the template. Alternatively, by specifying the *ControlIndex* parameter, a control can be referred to using its index in the dialog box template (0 is the first control in the template, 1 is the second, and so on).

Note: When *ControlIndex* is specified, **OptionGroup** statements do not count as a control.

If you hide the control that currently has the focus, BasicScript will automatically set focus to the next control in the tab order.

Picture Caching

When the dialog box is first created and before it is shown, BasicScript calls the dialog function with *action* set to 1. At this time, no pictures have been loaded into the picture controls contained in the dialog box template. After control returns from the dialog function and before the dialog box is shown, BasicScript will load the pictures of all visible picture controls. Thus, it is possible for the dialog function to hide certain picture controls, which prevents the associated pictures from being loaded and causes the dialog box to load faster. When a picture control is made visible for the first time, the associated picture will then be loaded.

Example

```
'This example creates a dialog box with two panels. The
'DlgVisible statement is used to show or hide the controls of
'the different panels.
Sub EnableGroup(start%, finish%)
   For i = 6 To 13              'Disable all options.
      DlgVisible i, False
   Next i
   For i = start% To finish%    'Enable only the right ones.
      DlgVisible i, True
   Next i
```

```
End Sub
Function DlgProc(ControlName$, Action%, SuppValue%)
   If Action% = 1 Then
      DlgValue "WhichOptions",0     'Set to save options.
      EnableGroup 6, 8              'Enable the save options.
   End If
   If Action% = 2 And ControlName$ = "SaveOptions" Then
      EnableGroup 6, 8              'Enable the save options.
      DlgProc = 1          'Don't close the dialog box.
   End If
   If Action% = 2 And ControlName$ = "EditingOptions" Then
      EnableGroup 9, 13    'Enable the editing options.
      DlgProc = 1          'Don't close the dialog box.
   End If
End Function
Sub Main()
   Begin Dialog OptionsDlg 33, 33, 171, 134, "Options", .DlgProc
      'Background (controls 0-5)
      GroupBox 8, 40, 152, 84, ""
      OptionGroup .WhichOptions
         OptionButton 8, 8, 59, 8, "Save Options",.SaveOptions
         OptionButton 8, 20, 65, 8, _
            "Editing Options",.EditingOptions
      OKButton 116, 7, 44, 14
      CancelButton 116, 24, 44, 14
      'Save options (controls 6-8)
      CheckBox 20, 56, 88, 8, "Always create backup",.CheckBox1
      CheckBox 20, 68, 65, 8, "Automatic save",.CheckBox2
      CheckBox 20, 80, 70, 8, "Allow overwriting",.CheckBox3
      'Editing options (controls 9-13)
      CheckBox 20, 56, 65, 8, "Overtype mode",.OvertypeMode
      CheckBox 20, 68, 69, 8, "Uppercase only",.UppercaseOnly
      CheckBox 20, 80, 105, 8, _
         "Automatically check syntax",.AutoCheckSyntax
      CheckBox 20, 92, 73, 8, _
         "Full line selection",.FullLineSelection
      CheckBox 20, 104, 102, 8, _
         "Typing replaces selection",.TypingReplacesText
   End Dialog
   Dim OptionsDialog As OptionsDlg
   Dialog OptionsDialog
End Sub
```

See Also DlgControlId (function); **DlgEnable** (function); **DlgEnable** (statement); **DlgFocus** (function); **DlgFocus** (statement); **DlgListBoxArray** (function); **DlgListBoxArray** (statement); **DlgSetPicture** (statement); **DlgText** (statement); **DlgText** (function); **DlgValue** (function); **DlgValue** (statement); **DlgVisible** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Do...Loop (statement)

Syntax 1
```
Do {While | Until} condition statements Loop
```

Syntax 2
```
Do
   statements
Loop {While | Until} condition
```

Syntax 3
```
Do
   statements
Loop
```

Description Repeats a block of BasicScript statements while a condition is **True** or until a condition is **True**.

Comments If the {**While** | **Until**} conditional clause is not specified, then the loop repeats the statements forever (or until BasicScript encounters an **Exit Do** statement).

The *condition* parameter specifies any **Boolean** expression.

Examples
```
Sub Main()
'This first example uses the Do...While statement, which
'performs the iteration, then checks the condition, and repeats
'if the condition is True.
   Dim a$(100)
   i% = -1
   Do
      i% = i% + 1
      If i% = 0 Then
         a(i%) = Dir$("*")
      Else
         a(i%) = Dir$
      End If
   Loop While (a(i%) <> "" And i% <= 99)
   r% = SelectBox(i% & " files found",,a)
'This second example uses the Do While...Loop, which checks the
'condition and then repeats if the condition is True.
   Dim a$(100)
   i% = 0
   a(i%) = Dir$("*")
   Do While a(i%) <> "" And i% <= 99
      i% = i% + 1
      a(i%) = Dir$
   Loop
   r% = SelectBox(i% & " files found",,a)
'This third example uses the Do Until...Loop, which does the
```

```
'iteration and then checks the condition and repeats if the
'condition is True.
   Dim a$(100)
   i% = 0
   a(i%) = Dir$("*")
   Do Until a(i%) = "" Or i% = 100
      i% = i% + 1
      a(i%) = Dir$
   Loop
   r% = SelectBox(i% & " files found",,a)
'This last example uses the Do...Until Loop, which performs the
'iteration first, checks the condition, and repeats if the
'condition is True.
   Dim a$(100)
   i% = -1
   Do
      i% = i% + 1
      If i% = 0 Then
         a(i%) = Dir$("*")
      Else
         a(i%) = Dir$
      End If
   Loop Until (a(i%) = "" Or i% = 100)
   r% = SelectBox(i% & " files found",,a)
End Sub
```

See Also **For...Next** (statement); **While...Wend** (statement).

Platform(s) All.

Platform Notes **Windows, Win32:** Due to errors in program logic, you can inadvertently create infinite loops in your code. Under Windows and Win 32, you can break out of infinite loops using Ctrl+Break.

UNIX: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under UNIX, you can break out of infinite loops using Ctrl+C.

Macintosh: Due to errors in program logic, you can inadvertently create infinite loops in your code. On the Macintosh, you can break out of infinite loops using Command+Period.

OS/2: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under OS/2, you can break out of infinite loops using Ctrl+C or Ctrl+Break.

DoEvents (function)

Syntax `DoEvents[()]`

Description Yields control to other applications, returning an **Integer** 0.

Comments This statement yields control to the operating system, allowing other applications to process mouse, keyboard, and other messages.

If a **SendKeys** statement is active, this statement waits until all the keys in the queue have been processed.

Example See **DoEvents** (statement).

See Also **DoEvents** (statement).

Platform(s) All.

Platform Notes **Win32:** Under Win32, this statement does nothing. Since Win32 systems are preemptive, use of this statement under these platforms is not necessary.

DoEvents (statement)

Syntax `DoEvents`

Description Yields control to other applications.

Comments This statement yields control to the operating system, allowing other applications to process mouse, keyboard, and other messages.

If a **SendKeys** statement is active, this statement waits until all the keys in the queue have been processed.

Examples

```
'This first example shows a script that takes a long time and
'hogs the system. The subroutine explicitly yields to allow
'other applications to execute.
Sub Main()
   Open "test.txt" For Output As #1
   For i = 1 To 10000
      Print #1,"This is a test of the system and stuff."
      DoEvents
   Next i
   Close #1
End Sub
'In this second example, the DoEvents statement is used to wait
'until the queue has been completely flushed.
Sub Main()
   AppActivate "Notepad"'Activate Notepad.
   SendKeys "This is a test.",False'Send some keys.
   DoEvents       'Wait for the keys to play back.
End Sub
```

See Also **DoEvents** (function).

Platform(s) All.

Platform Notes **Win32:** Under Win32, this statement does nothing. Since Win32 systems are preemptive, use of this statement under these platforms is not necessary.

DoKeys (statement)

Syntax `DoKeys` *KeyString$* `[,`*time*`]`

Description Simulates the pressing of the specified keys.

Comments The **DoKeys** statement accepts the following parameters:

Parameter	Description
KeyString$	**String** containing the keys to be sent. The format for *KeyString$* is described under the **SendKeys** statement.
time	**Integer** specifying the number of milliseconds devoted for the output of the entire *KeyString$* parameter. It must be within the following range: `0 <=` *time* `<= 32767` For example, if time is 5000 (5 seconds) and the *KeyString$* parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (or 0), the keys will play back at full speed.

Example

```
'This code fragment plays back the time and date into Notepad.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   id = Shell("Notepad",4)'Run Notepad.
   AppActivate "Notepad"
   t$ = time$
   d$ = date$
   DoKeys "The time is: " & t$ & "." & crlf
   DoKeys "The date is: " & d$ & "."
End Sub
```

See Also **SendKeys** (statement); **QueKeys** (statement); **QueKeyDn** (statement); **QueKeyUp** (statement).

Platform(s) Windows.

Platform Notes **Windows:** This statement uses the Windows journalizing mechanism to play keystrokes into the Windows environment.

Double (data type)

Syntax `Double`

Description A data type used to declare variables capable of holding real numbers with 15–16 digits of precision.

Comment Double variables are used to hold numbers within the following ranges:

Sign	Range
Negative	–1.797693134862315E308 <= *double* <= –4.94066E-324
Positive	4.94066E-324 <= *double* <= 1.797693134862315E308

The type-declaration character for **Double** is #.

Storage

Internally, doubles are 8-byte (64-bit) IEEE values. Thus, when appearing within a structure, doubles require 8 bytes of storage. When used with binary or random files, 8 bytes of storage are required.

Each **Double** consists of the following

- A 1-bit sign
- An 11-bit exponent
- A 53-bit significand (mantissa)

See Also **Currency** (data type); **Date** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CDbl** (function).

Platform(s) All.

DropListBox (statement)

Syntax `DropListBox` *x*, *y*, *width*, *height*, *ArrayVariable*, *.Identifier*

Description Creates a drop list box within a dialog box template.

Comments When the dialog box is invoked, the drop list box will be filled with the elements contained in *ArrayVariable*. Drop list boxes are similar to combo boxes, with the following exceptions:

- The list box portion of a drop list box is not opened by default. The user must open it by clicking the down arrow.

- The user cannot type into a drop list box. Only items from the list box may be selected. With combo boxes, the user can type the name of an item from the list directly or type the name of an item that is not contained within the combo box.

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **DropListBox** statement requires the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the dimensions of the control in dialog units.
ArrayVariable	Single-dimensioned array used to initialize the elements of the drop list box. If this array has no dimensions, then the drop list box will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. *ArrayVariable* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the index of the drop list box's selection (0 is the first item, 1 is the second, and so on). This variable can be accessed using the following syntax: *DialogVariable.Identifier*

Example

```
'This example allows the user to choose a field name from a drop
'list box.
Sub Main()
   Dim FieldNames$(4)
   FieldNames$(0) = "Last Name"
   FieldNames$(1) = "First Name"
   FieldNames$(2) = "Zip Code"
   FieldNames$(3) = "State"
   FieldNames$(4) = "City"
   Begin Dialog FindTemplate 16,32,168,48,"Find"
      Text 8,8,37,8,"&Find what:"
      DropListBox 48,6,64,80,FieldNames,.WhichField
      OKButton 120,7,40,14
      CancelButton 120,27,40,14
   End Dialog
   Dim FindDialog As FindTemplate
```

```
    FindDialog.WhichField = 1
    Dialog FindDialog
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

EditEnabled (function)

Syntax `EditEnabled(`*name$* | *id*`)`

Description Returns **True** if the given text box is enabled within the active window or dialog box; returns **False** otherwise.

Comments The **EditEnabled** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the text box. The name of a text box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a text box.
id	**Integer** specifying the ID of the text box.

A runtime error is generated if a text box control with the given name or ID cannot be found within the active window.

If enabled, the text box can be given the focus using the **ActivateControl** statement.

Note: The **EditEnabled** function is used to determine whether a text box is enabled in another application's dialog box. Use the **DlgEnable** function in dynamic dialog boxes.

Example

```
'This example adjusts the left margin if this control is enabled.
Sub Main()
   Menu "Format.Paragraph"
   If EditEnabled("Left:") Then
      SetEditText "Left:","5 pt"
   End If
End Sub
```

See Also **EditExists** (function); **GetEditText$** (function); **SetEditText** (statement).

Platform(s) Windows.

EditExists (function)

Syntax `EditExists(`*name$* | *id*`)`

Description Returns **True** if the given text box exists within the active window or dialog box; returns **False** otherwise.

Comments The **EditExists** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the text box. The name of a text box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a text box.
id	**Integer** specifying the ID of the text box.

A runtime error is generated if a text box control with the given name or ID cannot be found within the active window.

If there is no active window, **False** will be returned.

Note: The **EditExists** function is used to determine whether a text box exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This example adjusts the left margin if this control exists and
'is enabled.
Sub Main()
   Menu "Format.Paragraph"
   If EditExists("Left:") Then
      If EditEnabled("Left:") Then
         SetEditText "Left:","5 pt"
      End If
   End If
End Sub
```

See Also **EditEnabled** (function); **GetEditText$** (function); **SetEditText** (statement).

Platform(s) Windows.

End (statement)

Syntax

```
End
```

Description Terminates execution of the current script, closing all open files.

Example

```
'This example uses the End statement to stop execution.
Sub Main()
   MsgBox "The next line will terminate the script."
   End
End Sub
```

See Also **Close** (statement); **Stop** (statement); **Exit For** (statement); **Exit Do** (statement); **Exit Function** (statement); **Exit Sub** (statement).

Platform(s) All.

Environ, Environ$ (functions)

Syntax `Environ[$](`*variable$ | VariableNumber*`)`

Description Returns the value of the specified environment variable.

Comments **Environ$** returns a **String**, whereas **Environ** returns a **String** variant.

If *variable$* is specified, then this function looks for that *variable$* in the environment. If the *variable$* name cannot be found, then a zero-length string is returned.

If *VariableNumber* is specified, then this function looks for the *N*th variable within the environment (the first variable being number 1). If there is no such environment variable, then a zero-length string is returned. Otherwise, the entire entry from the environment is returned in the following format:

```
variable = value
```

Example

```
'This example looks for the DOS Comspec variable and displays
'the value in a dialog box.
Sub Main()
   Dim a$(1)
   a$(1) = Environ$("COMSPEC")
   MsgBox "The DOS Comspec variable is set to: " & a$(1)
End Sub
```

See Also **Command**, **Command$** (functions).

Platform(s) All.

EOF (function)

Syntax `EOF(`*filenumber*`)`

Description Returns **True** if the end-of-file has been reached for the given file; returns **False** otherwise.

Comments The *filenumber* parameter is an **Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.

With sequential files, **EOF** returns **True** when the end of the file has been reached (i.e., the next file read command will result in a runtime error).

With Random or Binary files, **EOF** returns **True** after an attempt has been made to read beyond the end of the file. Thus, **EOF** will only return **True** when **Get** was unable to read the entire record.

Example

```
'This example opens the autoexec.bat file and reads lines from
'the file until the end-of-file is reached.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim s$
   Open "c:\autoexec.bat" For Input As #1
   Do While Not EOF(1)
      Input #1,s$
   Loop
   Close
   MsgBox "The last line was:" & crlf & s$
End Sub
```

See Also **Open** (statement); **Lof** (function).

Platform(s) All.

Eqv (operator)

Syntax *result* = *expression1* Eqv *expression2*

Description Performs a logical or binary equivalence on two expressions.

Comments If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical equivalence is performed as follows:

If *expression1* **is**	**and** *expression2* **is**	**then the** *result* **is**
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

If either expression is **Null**, then **Null** is returned.

Binary Equivalence

If the two expressions are **Integer**, then a binary equivalence is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long** and a binary equivalence is then performed, returning a **Long** result.

Binary equivalence forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions, according to the following table:

If bit in *expression1* **is**	**and bit in** *expression2* **is**	**the** *result* **is**
1	1	1
0	1	0

If bit in *expression1* is	and bit in *expression2* is	the *result* is
1	0	0
0	0	1

Example

```
'This example assigns False to A, performs some equivalent
'operations, and displays a dialog box with the result. Since A
'is equivalent to False, and False is equivalent to 0, and by
'definition, A = 0, then the dialog box will display "A is False."
Sub Main()
   a = False
   If ((a Eqv False) And (False Eqv 0) And (a = 0)) Then
      MsgBox "a is False."
   Else
      MsgBox "a is True."
   End If
End Sub
```

See Also Operator Precedence (topic); **Or** (operator); **Xor** (operator); **Imp** (operator); **And** (operator).

Platform(s) All.

Erase (statement)

Syntax

```
Erase array1 [,array2]...
```

Description Erases the elements of the specified arrays.

Comments For dynamic arrays, the elements are erased, and the array is redimensioned to have no dimensions (and therefore no elements). For fixed arrays, only the elements are erased; the array dimensions are not changed.

After a dynamic array is erased, the array will contain no elements and no dimensions. Thus, before the array can be used by your program, the dimensions must be reestablished using the **Redim** statement.

Up to 32 parameters can be specified with the **Erase** statement.

The meaning of erasing an array element depends on the type of the element being erased:

Element Type	What Erase Does to That Element
Integer	Sets the element to 0.
Boolean	Sets the element to **False**.
Long	Sets the element to 0.

Element Type	What Erase Does to That Element
Double	Sets the element to 0.0.
Date	Sets the element to December 30, 1899.
Single	Sets the element to 0.0.
String (variable-length)	Frees the string, then sets the element to a zero-length string.
String (fixed-length)	Sets every character of each element to zero **(Chr$(0))**.
Object	Decrements the reference count and sets the element to **Nothing**.
Variant	Sets the element to **Empty**.
User-defined type	Sets each structure element as a separate variable.

Example

```
'This example puts a value into an array and displays it. Then
'it erases the value and displays it again.
Sub Main()
   Dim a$(10)  'Declare an array.
   a$(1) = Dir$("*")    'Fill element 1 with a filename.
   MsgBox "Array before Erase: " & a$(1)  'Display element 1.
   Erase a$       'Erase all elements in the array.
   MsgBox "Array after Erase: " & a$(1)   'Display element 1
                                          'again (should be
                                          'erased).
End Sub
```

See Also **Redim** (statement); Arrays (topic).

Platform(s) All.

Erl (function)

Syntax

```
Erl[()]
```

Description Returns the line number of the most recent error.

Comments The first line of the script is 1, the second line is 2, and so on.

The internal value of **Erl** is reset to 0 with any of the following statements: **Resume**, **Exit Sub**, **Exit Function**. Thus, if you want to use this value outside an error handler, you must assign it to a variable.

Example

```
'This example generates an error and then determines the line
'on which the error occurred.
Sub Main()
   Dim i As Integer
```

```
   On Error Goto Trap1
   i = 32767   'Generate an error--overflow.
   i = i + 1
   Exit Sub
Trap1:
   MsgBox "Error on line: " & Erl
   Exit Sub    'Reset the error handler.
End Sub
```

See Also Error Handling (topic).

Platform(s) All.

Err.Clear (method)

Syntax `Err.Clear`

Description Clears the properties of the **Err** object.

Comments After this method has been called, the properties of the **Err** object will have the following values:

Property	Value
Err.Description	""
Err.HelpContext	**0**
Err.HelpFile	""
Err.LastDLLError	**0**
Err.Number	**0**
Err.Source	""

The properties of the **Err** object are automatically reset when any of the following statements are executed:

Resume	Exit Function
On Error	Exit Sub

Example

```
'The following script gets input from the user using error
'checking.
Sub Main()
   Dim x As Integer
   On Error Resume Next
   x = InputBox("Type in a number")
   If Err.Number <> 0 Then
      Err.Clear
      x = 0
   End If
```

```
   MsgBox x
End Sub
```

See Also Error Handling (topic); **Err.Description** (property); **Err.HelpContext** (property); **Err.HelpFile** (property); **Err.LastDLLError** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) All.

Err.Description (property)

Syntax `Err.Description [= stringexpression]`

Description Sets or retrieves the description of the error.

Comments For errors generated by BasicScript, the **Err.Description** property is automatically set.

For user-defined errors, you should set this property to be a description of your error. If you set the **Err.Number** property to one of BasicScript's internal error numbers and you don't set the **Err.Description** property, then the **Err.Description** property is automatically set when the error is generated (i.e., with **Err.Raise**).

Example

```
'The following script gets input from the user using error
'checking. When an error occurs, the Err.Description property
'is displayed to the user and execution continues with a default
'value.
Sub Main()
   Dim x As Integer
   On Error Resume Next
   x = InputBox("Type in a number")
   If Err.Number <> 0 Then
      MsgBox "The following error occurred: " & Err.Description
      x = 0
   End If
   MsgBox x
End Sub
```

See Also Error Handling (topic); **Err.Clear** (method); **Err.HelpContext** (property); **Err.HelpFile** (property); **Err.LastDLLError** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) All.

Err.HelpContext (property)

Syntax `Err.HelpContext [= contextid]`

Description Sets or retrieves the help context ID that identifies the help topic for information on the error.

Comments The **Err.HelpContext** property, together with the **Err.HelpFile** property, contain sufficient information to display help for the error.

When BasicScript generates an error, the **Err.HelpContext** property is set to 0 and the **Err.HelpFile** property is set to ""; the value of the **Err.Number** property is sufficient for displaying help in this case. The exception is with errors generated by an OLE automation server; both the **Err.HelpFile** and **Err.HelpContext** properties are set by the server to values appropriate for the generated error.

When generating your own user-define errors, you should set the **Err.HelpContext** property and the **Err.HelpFile** property appropriately for your error. If these are not set, then BasicScript displays its own help at an appropriate place.

Example

```
'This example defines a replacement for InputBox that deals
'specifically with Integer values. If an error occurs, the
'function generates a user-defined error that can be trapped
'by the caller.
Function InputInteger(Prompt,Optional Title,Optional Def)
   On Error Resume Next
   Dim x As Integer
   x = InputBox(Prompt,Title,Def)
   If Err.Number Then
      Err.HelpFile = "AZ.HLP"
      Err.HelpContext = 2
      Err.Description = "Integer value expected"
      InputInteger = Null
      Err.Raise 3000
   End If
   InputInteger = x
End Function
Sub Main
   Dim x As Integer
   Do
      On Error Resume Next
      x = InputInteger("Enter a number:")
      If Err.Number = 3000 then
         Msgbox "You didn't type in a valid number, press ""F1"" _
               "to invoke help file."
      End If
   Loop Until Err.Number <> 3000
End Sub
```

See Also Error Handling (topic); **Err.Clear** (method); **Err.Description** (property); **Err.HelpFile** (property); **Err.LastDLLError** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) All.

Err.HelpFile (property)

Syntax `Err.HelpFile [= filename]`

Description Sets or retrieves the name of the help file associated with the error.

Comments The **Err.HelpFile** property, together with the **Err.HelpContents** property, contain sufficient information to display help for the error.

When BasicScript generates an error, the **Err.HelpContents** property is set to 0 and the **Err.HelpFile** property is set to ""; the value of the **Err.Number** property is sufficient for displaying help in this case. The exception is with errors generated by an OLE automation server; both the **Err.HelpFile** and **Err.HelpContext** properties are set by the server to values appropriate for the generated error.

When generating your own user-define errors, you should set the **Err.HelpContext** property and the **Err.HelpFile** property appropriately for your error. If these are not set, then BasicScript displays its own help at an appropriate place.

Example

```
'This example defines a replacement for InputBox that deals
'specifically with Integer values. If an error occurs, the
'function generates a user-defined error that can be trapped
'by the caller.
Function InputInteger(Prompt,Optional Title,Optional Def)
   On Error Resume Next
   Dim x As Integer
   x = InputBox(Prompt,Title,Def)
   If Err.Number Then
      Err.HelpFile = "AZ.HLP"
      Err.HelpContext = 2
      Err.Description = "Integer value expected"
      InputInteger = Null
      Err.Raise 3000
   End If
   InputInteger = x
End Function
Sub Main
   Dim x As Integer
   Do
      On Error Resume Next
      x = InputInteger("Enter a number:")
      If Err.Number = 3000 Then
         Msgbox "You didn't type in a valid number, press ""F1""_
               "to invoke helpfile."
      End If
```

```
   Loop Until Err.Number <> 3000
End Sub
```

See Also Error Handling (topic); **Err.Clear** (method); **Err.HelpContext** (property); **Err.Description** (property); **Err.LastDLLError** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) All.

Platform Notes **Windows, Win32:** On these platforms, the **Err.HelpFile** property can be set to any valid Windows help file (i.e., a file with a .HLP extension compatible with the WINHELP help engine).

Err.LastDLLError (property)

Syntax `Err.LastDLLError`

Description Returns the last error generated by an external call—i.e., a call to a routine declared with the **Declare** statement that resides in an external module.

Comments The **Err.LastDLLError** property is automatically set when calling a routine defined in an external module. If no error occurs within the external call, then this property will automatically be set to 0.

The **Err.LastDLLError** property will always return 0 on platform where this property is not supported.,

Example

```
'The following script calls the GetCurrentDirectoryA. If an
'error occurs, this Win32 function sets the Err.LastDLLError
'property which can be checked for.
Declare Sub GetCurrentDirectoryA Lib "kernel32" (ByVal DestLen _
   As Integer,ByVal lpDest As String)
Sub Main()
   Dim dest As String * 256
   Err.Clear
   GetCurrentDirectoryA len(dest),dest
   If Err.LastDLLError <> 0 Then
      MsgBox "Error " & Err.LastDLLError & " occurred."
   Else
      MsgBox "Current directory is " & dest
   End If
End Sub
```

See Also Error Handling (topic); **Err.Clear** (method); **Err.HelpContext** (property); **Err.Description** (property); **Err.HelpFile** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) Win32, OS/2.

Platform Notes **Win32:** On this platform, this property is set by DLL routines that set the last error using the Win32 function **SetLastError()**. BasicScript uses the Win32 function **GetLastError()** to retrieve the value of this property. The value 0 is returned when calling DLL routines that do not set an error.

Err.Number (property)

Syntax `Err.Number [= errornumber]`

Description Returns or sets the number of the error.

Comments The **Err.Number** property is set automatically when an error occurs. This property can be used within an error trap to determine which error occurred.

You can set the **Err.Number** property to any **Long** value.

The **Number** property is the default property of the **Err** object. This allows you to use older style syntax such as those shown below:

```
Err = 6
If Err = 6 Then MsgBox "Overflow"
```

The **Err** function can only be used while within an error trap.

The internal value of the **Err.Number** property is reset to 0 with any of the following statements: **Resume**, **Exit Sub**, **Exit Function**. Thus, if you want to use this value outside an error handler, you must assign it to a variable.

Setting **Err.Number** to –1 has the side effect of resetting the error state. This allows you to perform error trapping within an error handler. The ability to reset the error handler while within an error trap is not standard Basic. Normally, the error handler is reset only with the **Resume**, **Exit Sub**, **Exit Function**, **End Function**, or **End Sub** statements.

Example

```
'This example forces error 10, with a subsequent transfer to
'the TestError label. TestError tests the error and, if not
'error 55, resets Err to 999 (user-defined error) and returns
'to the Main subroutine.
Sub Main()
   On Error Goto TestError
   Error 10
   MsgBox "The returned error is: '" & Err() & " - " & _
      Error$ & "'"
   Exit Sub
TestError:
   If Err = 55 Then 'File already open.
      MsgBox "Cannot copy an open file. Close it and try again."
   Else
      MsgBox "Error '" & Err & "' has occurred!"
```

```
      Err = 999
   End If
   Resume Next
End Sub
```

See Also Error Handling (topic).

Platform(s) All.

Err.Raise (method)

Syntax `Err.Raise` *number* `[,[`*source*`] [,[`*description*`] [,[`*helpfile*`] [,`*helpcontext*`]]]]`

Description Generates a runtime error, setting the specified properties of the **Err** object.

Comments The **Err.Raise** method has the following named parameters:

Named Parameter	Description
number	A **Long** value indicating the error number to be generated. This parameter is required. Error predefined by BasicScript are in the range between 0 and 1000.
source	An optional **String** expression specifying the source of the error—i.e., the object or module that generated the error. If omitted, then BasicScript uses the name of the currently executing script.
description	An optional **String** expression describing the error. If omitted and *number* maps to a predefined BasicScript error number, then the corresponding predefined description is used. Otherwise, the error "Application-defined or object-define error" is used.
helpfile	An optional **String** expression specifying the name of the help file containing context-sensitive help for this error. If omitted and number maps to a predefined BasicScript error number, then the default help file is assumed.
helpcontext	An optional Long value specifying the topic within *helpfile* containing context-sensitive help for this error.

If some arguments are omitted, then the current property values of the **Err** object are used.

This method can be used in place of the Error statement for generating errors. Using the **Err.Raise** method gives you the opportunity to set the desired properties of the **Err** object in one statement.

Example

```
'The following example uses the Err.Raise method to generate
'a user-defined error.
Sub Main()
   Dim x As Variant
   On Error Goto TRAP
   x = InputBox("Enter a number:")
   If Not IsNumber(x) Then
      Err.Raise 3000,,"Invalid number specified","WIDGET.HLP",30
   End If
   MsgBox x
   Exit Sub
TRAP:
   MsgBox Err.Description
End Sub
```

See Also **Error** (statement); Error Handling (topic); **Err.Clear** (method); **Err.HelpContext** (property); **Err.Description** (property); **Err.HelpFile** (property); **Err.Number** (property); **Err.Source** (property).

Platform(s) All.

Err.Source (property)

Syntax `Err.Source [= stringexpression]`

Description Sets or retrieves the source of a runtime error.

Comments For OLE automation errors generated by the OLE server, the **Err.Source** property is set to the name of the object that generated the error. For all other errors generated by BasicScript, the **Err.Source** property is automatically set to be the name of the script that generated the error.

For user-defined errors, the **Err.Source** property can be set to any valid String expression indicating the source of the error. If the **Err.Source** property is not explicitly set for user-defined errors, the BasicScript sets the value to be the name of the script in which the error was generated.

Example

```
'The following script generates an error, setting the source
'to the specific location where the error was generated.
Function InputInteger(Prompt,Optional Title,Optional Def)
   On Error Resume Next
   Dim x As Integer
   x = InputBox(Prompt,Title,Def)
   If Err.Number Then
```

```
      Err.Source = "InputInteger"
      Err.Description = "Integer value expected"
      InputInteger = Null
      Err.Raise 3000
   End If
   InputInteger = x
End Function
Sub Main
   On Error Resume Next
   x = InputInteger("Enter a number:")
   If Err.Number Then MsgBox Err.Source & ":" & Err.Description
End Sub
```

See Also Error Handling (topic); **Err.Clear** (method); **Err.HelpContext** (property); **Err.Description** (property); **Err.HelpFile** (property); **Err.Number** (property); **Err.LastDLLError** (property).

Platform(s) All.

Error (statement)

Syntax `Error` *errornumber*

Description Simulates the occurrence of the given runtime error.

Comments The *errornumber* parameter is any **Integer** containing either a built-in error number or a user-defined error number. The **Err.Number** property can be used within the error trap handler to determine the value of the error.

The **Error** statement is provided for backward compatibility. Use the **Err.Raise** method instead. When using the **Error** statement to generate an error, the **Err** object's properties are set to the following default values:

Property	Default Value
Number	This property is set to *errornumber* as specified in the **Error** statement.
Source	Name of the currently executing script.
Description	Text of the error. If *errornumber* does not specify a known BasicScript error, then **Description** is set to an empty string.
HelpFile	Name of the BasicScript help file.
HelpContext	Context ID corresponding to *errornumber*.

A runtime error is generated if *errornumber* is less than 0.

Example

```
'This example forces error 10, with a subsequent transfer to
'the TestError label. TestError tests the error and, if not
```

```
'error 55, resets Err to 999 (user-defined error) and returns
'to the Main subroutine.
Sub Main()
   On Error Goto TestError
   Error 10
   MsgBox "The returned error is: '" & Err & " - " & Error$ & "'"
   Exit Sub
TestError:
   If Err = 55 Then 'File already open.
      MsgBox "Cannot copy an open file. Close it and try again."
   Else
      MsgBox "Error '" & Err & "' has occurred."
      Err = 999
   End If
   Resume Next
End Sub
```

See Also Error Handling (topic).

Platform(s) All.

Error Handling (topic)

Error Handlers

BasicScript supports nested error handlers. When an error occurs within a subroutine, BasicScript checks for an **On Error** handler within the currently executing subroutine or function. An error handler is defined as follows:

```
Sub foo()
   On Error Goto catch
   'Do something here.
   Exit Sub
catch:
   'Handle error here.
End Sub
```

Error handlers have a life local to the procedure in which they are defined. The error is reset when any of the following conditions occurs:

- An **On Error** or **Resume** statement is encountered.
- When **Err.Number** is set to -1.
- When the **Err.Clear** method is called.
- When an **Exit Sub**, **Exit Function**, **End Function**, **End Sub** is encountered.

Cascading Errors

If a runtime error occurs and no **On Error** handler is defined within the currently executing procedure, then BasicScript returns to the calling procedure and executes the error handler there. This process repeats until a procedure is found that contains an error handler or until there are no more procedures. If an error is not trapped or if an error occurs within the error handler, then BasicScript displays an error message, halting execution of the script.

Once an error handler has control, it should address the condition that caused the error and resume execution with the **Resume** statement. This statement resets the error handler, transferring execution to an appropriate place within the current procedure. The error is reset if the procedure exits without first executing **Resume**.

Visual Basic Compatibility

Where possible, BasicScript has the same error numbers and error messages as Visual Basic. This is useful for porting scripts between environments.

Handling errors in BasicScript involves querying the error number or error text using the **Error$** function or **Err.Description** property. Since this is the only way to handle errors in BasicScript, compatibility with Visual Basic's error numbers and messages is essential.

BasicScript errors fall into three categories:

1. **Visual Basic-compatible errors:** These errors, numbered between 0 and 799, are numbered and named according to the errors supported by Visual Basic.
2. **BasicScript errors:** These errors, numbered from 800 to 999, are unique to BasicScript.
3. **User-defined errors:** These errors, equal to or greater than 1,000, are available for use by extensions or by the script itself.

You can intercept trappable errors using BasicScript's **On Error** construct. Almost all errors in BasicScript are trappable except for various system errors.

Error, Error$ (functions)

Syntax `Error[$][(`*errornumber*`)]`

Description Returns a **String** containing the text corresponding to the given error number or the most recent error.

Comments **Error$** returns a **String**, whereas **Error** returns a **String** variant.

The *errornumber* parameter is an **Integer** containing the number of the error message to retrieve. If this parameter is omitted, then the function returns the text corresponding to the most recent runtime error (i.e., the same as returned by the **Err.Description** property). If no runtime error has occurred, then a zero-length string is returned.

If the **Error** statement was used to generate a user-defined runtime error, then this function will return a zero-length string ("").

Example

```
'This example forces error 10, with a subsequent transfer to
'the TestError label. TestError tests the error and, if not
'error 55, resets Err to 999 (user-defined error) and returns
'to the Main subroutine.
Sub Main()
   On Error Goto TestError
   Error 10
   MsgBox "The returned error is: '" & Err() & " - " _
      & Error$ & "'"
   Exit Sub
TestError:
   If Err = 55 Then 'File already open.
      MsgBox "Cannot copy an open file. Close it and try again."
   Else
      MsgBox "Error '" & Err & "' has occurred."
      Err = 999
   End If
   Resume Next
End Sub
```

See Also Error Handling (topic).

Platform(s) All.

Exit Do (statement)

Syntax

```
Exit Do
```

Description Causes execution to continue on the statement following the **Loop** clause.

Comments This statement can only appear within a **Do...Loop** statement.

Example

```
'This example will load an array with directory entries unless
'there are more than ten entries--in which case, the Exit Do
'terminates the loop.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim a$(5)
   Do
      i% = i% + 1
```

```
      If i% = 1 Then
         a(i%) = Dir$("*")
      Else
         a(i%) = Dir$
      End If
      If i% >= 10 Then Exit Do
   Loop While (a(i%) <> "")
   If i% = 10 Then
      MsgBox i% & " entries processed!"
   Else
      MsgBox "Less than " & i% & " entries processed!"
   End If
End Sub
```

See Also **Stop** (statement); **Exit For** (statement); **Exit Function** (statement); **Exit Sub** (statement); **End** (statement); **Do...Loop** (statement).

Platform(s) All.

Exit For (statement)

Syntax

```
Exit For
```

Description Causes execution to exit the innermost **For** loop, continuing execution on the line following the **Next** statement.

Comments This statement can only appear within a **For...Next** block.

Example

```
'This example will fill an array with directory entries until a
'null entry is encountered or 100 entries have been processed--
'at which time, the loop is terminated by an Exit For statement.
'The dialog box displays a count of files found and then some
'entries from the array.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim a$(100)
   For i = 1 To 100
      If i = 1 Then
         a$(i) = Dir$("*")
      Else
         a$(i) = Dir$
      End If
      If (a$(i) = "") Or (i >= 100) Then Exit For
   Next i
   message = "There are " & i & " files found." & crlf
   MsgBox message & a$(1) & crlf & a$(2) & crlf & a$(3) & crlf &
a$(10)
End Sub
```

See Also **Stop** (statement); **Exit Do** (statement); **Exit Function** (statement); **Exit Sub** (statement); **End** (statement); **For...Next** (statement).

Platform(s) All.

Exit Function (statement)

Syntax
```
Exit Function
```

Description Causes execution to exit the current function, continuing execution on the statement following the call to this function.

Comments This statement can only appear within a function.

Example
```
'This function displays a message and then terminates with Exit
'Function.
Function Test_Exit() As Integer
   MsgBox "Testing function exit, returning to Main()."
   Test_Exit = 0
   Exit Function
   MsgBox "This line should never execute."
End Function
Sub Main()
   a% = Test_Exit()
   MsgBox "This is the last line of Main()."
End Sub
```

See Also **Stop** (statement); **Exit For** (statement); **Exit Do** (statement); **Exit Sub** (statement); **End** (statement); **Function...End Function** (statement).

Platform(s) All.

Exit Sub (statement)

Syntax
```
Exit Sub
```

Description Causes execution to exit the current subroutine, continuing execution on the statement following the call to this subroutine.

Comments This statement can appear anywhere within a subroutine. It cannot appear within a function.

Example
```
'This example displays a dialog box and then exits. The last
'line should never execute because of the Exit Sub statement.
Sub Main()
   MsgBox "Terminating Main()."
   Exit Sub
```

```
   MsgBox "Still here in Main()."
End Sub
```

See Also **Stop** (statement); **Exit For** (statement); **Exit Do** (statement); **Exit Function** (statement); **End** (function); **Sub...End Sub** (statement).

Platform(s) All.

Exp (function)

Syntax `Exp(`*number*`)`

Description Returns the value of *e* raised to the power of *number*.

Comments The *number* parameter is a **Double** within the following range:

```
0 <= number <= 709.782712893.
```

A runtime error is generated if *number* is out of the range specified above.

The value of *e* is 2.71828.

Example

```
'This example assigns a to e raised to the 12.4 power and
'displays it in a dialog box.
Sub Main()
   a# = Exp(12.40)
   MsgBox "e to the 12.4 power is: " & a#
End Sub
```

See Also **Log** (function).

Platform(s) All.

Expression Evaluation (topic)

BasicScript allows expressions to involve data of different types. When this occurs, the two arguments are converted to be of the same type by promoting the less precise operand to the same type as the more precise operand. For example, BasicScript will promote the value of i% to a Double in the following expression:

```
result# = i% * d#
```

In some cases, the data type to which each operand is promoted is different than that of the most precise operand. This is dependent on the operator and the data types of the two operands and is noted in the description of each operator.

If an operation is performed between a numeric expression and a **String** expression, then the **String** expression is usually converted to be of the same type as the numeric expression. For example, the following expression converts the **String** expression to an **Integer** before performing the multiplication:

```
result = 10 * "2" 'Result is equal to 20.
```

There are exceptions to this rule, as noted in the description of the individual operators.

Type Coercion

BasicScript performs numeric type conversion automatically. Automatic conversions sometimes result in overflow errors, as shown in the following example:

```
d# = 45354
i% = d#
```

In this example, an overflow error is generated because the value contained in d# is larger than the maximum size of an **Integer**.

Rounding

When floating-point values (**Single** or **Double**) are converted to integer values (**Integer** or **Long**), the fractional part of the floating-point number is lost, rounding to the nearest integer value. BasicScript uses Baker's rounding:

- If the fractional part is larger than .5, the number is rounded up.
- If the fractional part is smaller than .5, the number is rounded down.
- If the fractional part is equal to .5, then the number is rounded up if it is odd and down if it is even.

The following table shows sample values before and after rounding:

Before Rounding	After Rounding to Whole Number
2.1	2
4.6	5
2.5	2
3.5	4

Default Properties

When an OLE object variable or an **Object** variant is used with numerical operators such as addition or subtraction, then the default property of that object is automatically retrieved. For example, consider the following:

```
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")
MsgBox "This application is " & Excel
```

The above example displays "This application is Microsoft Excel" in a dialog box. When the variable Excel is used within the expression, the default property is automatically retrieved, which, in this case, is the string "Microsoft Excel." Considering that the default property of the Excel object is .Value, then the following two statements are equivalent:

```
MsgBox "This application is " & Excel
MsgBox "This application is " & Excel.Value
```

FileAttr (function)

Syntax `FileAttr(`*filenumber*`, `*returntype*`)`

Description Returns an **Integer** specifying the file mode (if *returntype* is 1) or the operating system file handle (if *returntype* is 2).

Comments The **FileAttr** function takes the following named parameters:

Named Parameter	Description
filenumber	**Integer** value used by BasicScript to refer to the open file—the number passed to the **Open** statement.
returntype	**Integer** specifying the type of value to be returned. If *returntype* is 1, then one of the following values is returned: 1 **Input** **2 Output** 4 **Random** 6 **Append** 32 **Binary** If *returntype* is 2, then the operating system file handle is returned. On most systems, this is a special **Integer** value identifying the file.

Example

```
'This example opens a file for input, reads the file attributes,
'and determines the file mode for which it was opened. The
'result is displayed in a dialog box.
Sub Main()
   Open "c:\autoexec.bat" For Input As #1
   a% = FileAttr(1,1)
   Select Case a%
      Case 1
         MsgBox "Opened for input."
      Case 2
         MsgBox "Opened for output."
      Case 4
         MsgBox "Opened for random."
      Case 8
         MsgBox "Opened for append."
      Case 32
         MsgBox "Opened for binary."
      Case Else
         MsgBox "Unknown file mode."
      End Select
   a% = FileAttr(1,2)
```

```
   MsgBox "File handle is: " & a%
   Close
End Sub
```

See Also **FileLen** (function); **GetAttr** (function); **FileType** (function); **FileExists** (function); **Open** (statement); **SetAttr** (statement).

Platform(s) All.

FileCopy (statement)

Syntax `FileCopy` *source*, *destination*

Description Copies a *source* file to a *destination* file.

Comments The **FileCopy** function takes the following named parameters:

Named Parameter	Description
source	**String** containing the name of a single file to copy. The *source* parameter cannot contain wildcards (**?** or *****) but may contain path information.
destination	**String** containing a single, unique destination file, which may contain a drive and path specification.

The file will be copied and renamed if the *source* and *destination* filenames are not the same.

Some platforms do not support drive letters and may not support dots to indicate current and parent directories.

Example

```
'This example copies the autoexec.bat file to "autoexec.sav",
'then opens the copied file and tries to copy it again--which
'generates an error.
Sub Main()
   On Error Goto ErrHandler
   FileCopy "c:\autoexec.bat", "c:\autoexec.sav"
   Open "c:\autoexec.sav" For Input As # 1
   FileCopy "c:\autoexec.sav", "c:\autoexec.sv2"
   Close
   Exit Sub
ErrHandler:
   If Err = 55 Then 'File already open.
      MsgBox "Cannot copy an open file. Close it and try again."
   Else
      MsgBox "An unspecified file copy error has occurred."
   End If
   Resume Next
```

```
End Sub
```

See Also **Kill** (statement); **Name** (statement).

Platform(s) All.

FileDateTime (function)

Syntax `FileDateTime(`*pathname*`)`

Description Returns a **Date** variant representing the date and time of the last modification of a file.

Comments This function retrieves the date and time of the last modification of the file specified by *pathname* (wildcards are not allowed). A runtime error results if the file does not exist. The value returned can be used with the date/time functions (i.e., **Year**, **Month**, **Day**, **Weekday**, **Minute**, **Second**, **Hour**) to extract the individual elements.

Some operating systems (such as Win32) store the file creation date, last modification date, and the date the file was last written to. The **FileDateTime** function only returns the last modification date.

Example

```
'This example gets the file date/time of the autoexec.bat file
'and displays it in a dialog box.
Sub Main()
   If FileExists("c:\autoexec.bat") Then
      a# = FileDateTime("c:\autoexec.bat")
      MsgBox "The date/time information for the file is: " _
         & Year(a#) & "-" & Month(a#) & "-" & Day(a#)
   Else
      MsgBox "The file does not exist."
   End If
End Sub
```

See Also **FileLen** (function); **GetAttr** (function); **FileType** (function); **FileAttr** (function); **FileExists** (function).

Platform(s) All.

FileDirs (statement)

Syntax `FileDirs` *array()* `[,`*dirspec$*`]`

Description Fills a **String** or **Variant** array with directory names from disk.

Comments The **FileDirs** statement takes the following parameters:

Parameter	Description
array()	Either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If *array()* is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions. If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for **String** arrays) or **Empty** (for **Variant** arrays). A runtime error results if the array is too small to hold the new elements.

dirspec$	String containing the file search mask, such as:
	`t*.`
	`c:\*.*`
	
	If this parameter is omitted or an empty string, then ***** is used, which fills the array with all the subdirectory names within the current directory.

Example

```
'This example fills an array with directory entries and displays
'the first one.
Sub Main()
   Dim a$()
   FileDirs a$,"c:\*.*"
   MsgBox "The first directory is: " & a$(0)
End Sub
```

See Also **FileList** (statement); **Dir**, **Dir$** (functions); **CurDir**, **CurDir$** (functions); **ChDir** (statement).

Platform(s) All.

FileExists (function)

Syntax `FileExists(`*filename$*`)`

Description Returns **True** if *filename$* exists; returns **False** otherwise.

Comments This function determines whether a given *filename$* is valid.

This function will return **False** if *filename$* specifies a subdirectory.

Note: On some file systems, the directories "." and ".." will be returned.

Example

```
'This example checks to see whether there is an autoexec.bat
'file in the root directory of the C drive, then displays either
'its date and time of creation or the fact that it does not exist.
Sub Main()
   If FileExists("c:\autoexec.bat") Then
      Msgbox "This file exists!"
   Else
      MsgBox "File does not exist."
   End If
End Sub
```

See Also **FileLen** (function); **GetAttr** (function); **FileType** (function); **FileAttr** (function); **FileParse$** (function).

Platform(s) All.

FileLen (function)

Syntax `FileLen(`*pathname*`)`

Description Returns a **Long** representing the length of *pathname* in bytes.

Comments This function is used in place of the **LOF** function to retrieve the length of a file without first opening the file. A runtime error results if the file does not exist.

Example

```
'This example checks to see whether there is a c:\autoexec.bat
'file and, if there is, displays the length of the file.
Sub Main()
   If (FileExists("c:\autoexec.bat") _
      And (FileLen("c:\autoexec.bat") <> 0)) Then
         b% = FileLen("c:\autoexec.bat")
         MsgBox "The length of autoexec.bat is: " & b%
   Else
      MsgBox "File does not exist."
   End If
End Sub
```

See Also **GetAttr** (function); **FileType** (function); **FileAttr** (function); **FileParse$** (function); **FileExists** (function); **Loc** (function).

Platform(s) All.

FileList (statement)

Syntax `FileList` *array()* `[,[`*filespec$*`] [,[`*include_attr*`] [,`*exclude_attr*`]]]`

Description Fills a **String** or **Variant** array with filenames from disk.

Comments The **FileList** function takes the following parameters:

Parameter	Description
array()	Either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If *array()* is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions f the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for **String** arrays) or **Empty** (for **Variant** arrays). A runtime error results if the array is too small to hold the new elements.
filespec$	String specifying which filenames are to be included in the list. The *filespec$* parameter can include wildcards, such as ***** and **?**. If this parameter is omitted, then ***** is used.
include_attr	**Integer** specifying attributes of files you want included in the list. It can be any combination of the attributes listed below.
exclude_attr	**Integer** specifying attributes of files you want excluded from the list. It can be any combination of the attributes listed below.

The **FileList** function returns different files as specified by the *include_attr* and *exclude_attr* and whether these parameter have been specified. The following table shows these differences:If neither the include_attr or exclude_attr have been specified, then the following defaults are assumed:

Parameter	Default
exclude_attr	ebHidden Or ebDirectory Or ebSystem Or ebVolume
include_attr	ebNone Or ebArchive Or ebReadOnly

If *include_attr* is specified and *exclude_attr* is missing, then **FileList** excludes all files not specified by *include_attr*. If *include_attr* is missing, its value is assumed to be zero.

Wildcards

The ***** character matches any sequence of zero or more characters, whereas the **?** character matches any single character. Multiple *****'s and **?**'s can appear within the expression to form complete searching patterns. The following table shows some examples:

This pattern	Matches these files	Doesn't match these files
*S.*TXT	SAMPLE. TXT	SAMPLE
	GOOSE.TXT	SAMPLE.DAT
	SAMS.TXT	
C*T.TXT	CAT.TXT	CAP.TXT
		ACATS.TXT
C*T	CAT	CAT.DOC
	CAP.TXT	
C?T	CAT	CAT.TXT
	CUT	CAPIT
		CT
*	(All files)	

File Attributes

These numbers can be any combination of the following:

Constant	Value	Includes
ebNormal	0	Read-only, archive, subdir, none
ebReadOnly	1	Read-only files
ebHidden	2	Hidden files
ebSystem	4	System files
ebVolume	8	Volume label
ebDirectory	16	Subdirectories
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

Example

```
'This example fills an array a with the directory of the current
'drive for all files that have normal or no attributes and
'excludes those with system attributes. The dialog box displays
'four filenames from the array.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
```

```
   Dim a$()
   FileList a$,"*.*", (ebNormal + ebNone), ebSystem
   If ArrayDims(a$) > 0 Then
      MsgBox a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(4)
   Else
      MsgBox "No files found."
   End If
End Sub
```

See Also **FileDirs** (statement); **Dir**, **Dir$** (functions).

Platform(s) All.

Platform Notes **Windows:** For compatibility with DOS wildcard matching, BasicScript special-cases the pattern "*.*" to indicate all files, not just files with a periods in their names.

UNIX: On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

FileParse$ (function)

Syntax `FileParse$(`*filename$*`[, `*operation*`])`

Description Returns a **String** containing a portion of *filename$* such as the path, drive, or file extension.

Comments The *filename$* parameter can specify any valid filename (it does not have to exist). For example:

```
..\test.dat
c:\sheets\test.dat
test.dat
```

A runtime error is generated if *filename$* is a zero-length string.

The optional *operation* parameter is an **Integer** specifying which portion of the *filename$* to extract. It can be any of the following values.

Value	Meaning	Example
0	Full name	`c:\sheets\test.dat`
1	Drive	`c`
2	Path	`c:\sheets`
3	Name	`test.dat`
4	Root	`test`
5	Extension	`dat`

If *operation* is not specified, then the full name is returned. A runtime error will result if *operation* is not one of the above values.

A runtime error results if *filename$* is empty.

On systems that do not support drive letters, operation 1 will return a zero-length string.

Example

```
'This example parses the file string "c:\testsub\autoexec.bat"
'into its component parts and displays them in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim a$(6)
   For i = 1 To 5
      a$(i) = FileParse$("c:\testsub\autoexec.bat",i - 1)
   Next i
   MsgBox a$(1) & crlf & a$(2) & crlf & a$(3) & crlf & a$(4) &
crlf & a$(5)
End Sub
```

See Also **FileLen** (function); **GetAttr** (function); **FileType** (function); **FileAttr** (function); **FileExists** (function).

Platform(s) All.

Platform Notes **Windows, Win32, OS/2:** The path separator is different on different platforms. Under Windows, OS/2, and Win32, the backslash and forward slash can be used interchangeably. For example, "c:\test.dat" is the same as "c:/test.dat".

UNIX: Under UNIX systems, the backslash and colon are valid filename characters.

Macintosh: On the Macintosh, all characters are valid within filenames except colons, which are seen as path separators.

NetWare: Under NetWare, operation 1 returns the volume name (up to 14 characters).

FileType (function)

Syntax `FileType(`*filename*`$)`

Description Returns the type of the specified file.

Comments One of the following **Integer** constants is returned:

Constant	Value	Description
ebDos	1	DOS executable file(exe files only; com files are not recognized).
ebWindows	2	Windows executable file

If one of the above values is not returned, then the file type is unknown.

Example

```
'This example looks at c:\windows\winfile.exe and determines
'whether it is a DOS or a Windows file. The result is displayed
'in a dialog box.
Sub Main()
   a = FileType("c:\windows\winfile.exe")
   If a = ebDos Then
      MsgBox "This is a DOS file."
   Else
      MsgBox "This is a Windows file of type '" & a & "'"
   End If
End Sub
```

See Also **FileLen** (function); **GetAttr** (function); **FileAttr** (function); **FileExists** (function).

Platform(s) Windows.

Platform Notes **Windows:** Only files with a ".exe" extension can be used with this function. Files with a ".com" or ".bat" extension will return 3 (unknown).

Fix (function)

Syntax `Fix(`*number*`)`

Description Returns the integer part of *number*.

Comments This function returns the integer part of the given value by removing the fractional part. The sign is preserved.

The **Fix** function returns the same type as *number*, with the following exceptions:

- If *number* is **Empty**, then an **Integer** variant of value 0 is returned.
- If *number* is a **String**, then a **Double** variant is returned.
- If *number* contains no valid data, then a **Null** variant is returned.

Example

```
'This example returns the fixed part of a number and assigns it
'to b, then displays the result in a dialog box.
Sub Main()
   a# = -19923.45
   b% = Fix(a#)
   MsgBox "The fixed portion of -19923.45 is: " & b%
End Sub
```

See Also **Int** (function); **CInt** (function).

Platform(s) All.

For Each...Next (statement)

Syntax

```
For Each member in group
   [statements]
   [Exit For]
   [statements]
Next [member]
```

Description

Repeats a block of statements for each element in a collection or array.

Comments

The **For Each...Next** statement takes the following parameters:

Parameter	Description
member	Name of the variable used for each iteration of the loop. If *group* is an array, then *member* must be a **Variant** variable. If *group* is a collection, then *member* must be an **Object** variable, an explicit OLE automation object, or a **Variant**.
group	Name of a collection or array.
statements	Any number of BasicScript statements.

BasicScript supports iteration through the elements of OLE collections or arrays, unless the arrays contain user-defined types or fixed-length strings. The iteration variable is a copy of the collection or array element in the sense thata change to the value of *member* within the loop has no effect on the collection or array.

The **For Each...Next** statement traverses array elements in the same order the elements are stored in memory. For example, the array elements contained in the array defined by the statement

Dim a(1 To 2,3 To 4)

are traversed in the following order: (1,3), (1,4), (2,3), (2,4). The order in which the elements are traversed should not be relevant to the correct operation of the script.

The **For Each...Next** statement continues executing until there are no more elements in *group* or until an **Exit For** statement is encountered.

For Each...Next statements can be nested. In such a case, the **Next** [*member*] statement applies to the innermost **For Each...Next** or **For...Next** statement. Each *member* variable of nested **For Each...Next** statements must be unique.

A **Next** statement appearing by itself (with no *member* variable) matches the innermost **For Each...Next** or **For...Next** loop.

Example

```
'The following subroutine iterates through the elements
'of an array using For Each...Next.
Sub Main()
   Dim a(3 To 10) As Single
   Dim i As Variant
```

```
   Dim s As String
   For i = 3 To 10
      a(i) = Rnd()
   Next i
   For Each i In a
      i = i + 1
   Next i
   s = ""
   For Each i In a
      If s <> "" Then s = s & ","
      s = s & i
   Next i
   MsgBox s
End Sub
'The following subroutine displays the names of each worksheet
'in an Excel workbook.
Sub Main()
   Dim Excel As Object
   Dim Sheets As Object
   Set Excel = CreateObject("Excel.Application")
   Excel.Visible = 1
   Excel.Workbooks.Add
   Set Sheets = Excel.Worksheets
   For Each a In Sheets
      MsgBox a.Name
   Next a
End Sub
```

See Also **Do...Loop** (statement); **While...Wend** (statement); **For...Next** (statement).

Platform(s) All.

Platform Notes **Windows, Win32:** Due to errors in program logic, you can inadvertently create infinite loops in your code. Under Windows and Win32, you can break out of infinite loops using Ctrl+Break.

UNIX: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under UNIX, you can break out of infinite loops using Ctrl+C.

Macintosh: Due to errors in program logic, you can inadvertently create infinite loops in your code. On the Macintosh, you can break out of infinite loops using Command+Period.

OS/2: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under OS/2, you can break out of infinite loops using Ctrl+C or Ctrl+Break.

For...Next (statement)

Syntax

```
For counter = start To end [Step increment]
   [statements]
   [Exit For]
   [statements]
Next [counter [,nextcounter]... ]
```

Description

Repeats a block of statements a specified number of times, incrementing a loop counter by a given increment each time through the loop.

Comments

The *For* statement takes the following parameters:

Parameter	Description
counter	Name of a numeric variable. Variables of the following types can be used: **Integer**, **Long**, **Single**, **Double**, **Variant**.
start	Initial value for *counter*. The first time through the loop, *counter* is assigned this value.
end	Final value for *counter*. The *statements* will continue executing until *counter* is equal to *end*.
increment	Amount added to counter each time through the loop. If *end* is greater than *start*, then *increment* must be positive. If *end* is less than *start*, then *increment* must be negative. If *increment* is not specified, then 1 is assumed. The expression given as *increment* is evaluated only once. Changing the step during execution of the loop will have no effect.
statements	Any number of BasicScript statements.

The **For...Next** statement continues executing until an **Exit For** statement is encountered when *counter* is greater than *end*.

For...Next statements can be nested. In such a case, the **Next** [*counter*] statement applies to the innermost **For...Next**.

The **Next** clause can be optimized for nested next loops by separating each counter with a comma. The ordering of the counters must be consistent with the nesting order (innermost counter appearing before outermost counter). The following example shows two equivalent **For** statements:

```
For i = 1 To 10                 For i = 1 To 10
   For j = 1 To 10              For j = 1 To 10
   Next j                       Next j,i
Next i
```

A **Next** clause appearing by itself (with no *counter* variable) matches the innermost **For** loop.

The *counter* variable can be changed within the loop but will have no effect on the number of times the loop will execute.

Example

```
'This example constructs a truth table for the OR statement
'using nested For...Next loops.
Sub Main()
   Dim m As String
   For x = -1 To 0
      For y = -1 To 0
         z = x Or y
         m = m & Format(Abs(x),"0") & " Or "
         m = m & Format(Abs(y),"0") & " = "
         m = m & Format(Z,"True/False") & Basic.Eoln$
      Next y
   Next x
   MsgBox m
End Sub
```

See Also **Do...Loop** (statement); **While...Wend** (statement); **For...Each** (statement).

Platform(s) All.

Platform Notes **Windows, Win32:** Due to errors in program logic, you can inadvertently create infinite loops in your code. Under Windows and Win32, you can break out of infinite loops using Ctrl+Break.

UNIX: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under UNIX, you can break out of infinite loops using Ctrl+C.

Macintosh: Due to errors in program logic, you can inadvertently create infinite loops in your code. On the Macintosh, you can break out of infinite loops using Command+Period.

OS/2: Due to errors in program logic, you can inadvertently create infinite loops in your code. Under OS/2, you can break out of infinite loops using Ctrl+C or Ctrl+Break.

Format, Format$ (functions)

Syntax `Format[$](`*expression* `[, [`*format*`] [, [`*firstdayofweek*`] [, `*firstweekofyear*`]])`

Description Returns a **String** formatted to user specification.

Comments **Format$** returns a **String**, whereas **Format** returns a **String** variant.

The **Format$**/**Format** functions take the following named parameters:

Named Parameter	Description
expression	String or numeric expression to be formatted. BasicScript will only examine the first 255 characters of *expression*.
format	Format expression that can be either one of the built-in BasicScript formats or a user-defined format consisting of characters that specify how the expression should be displayed. String, numeric, and date/time formats cannot be mixed in a single *format* expression.
firstdayofweek	Indicates the first day of the week. If omitted, then sunday is assumed (i.e., the constant **ebSunday** described below).
firstweekofyear	Indicates the first week of the year. If omitted, then the first week of the year is considered to be that containing January 1 (i.e., the constant **ebFirstJan1** as described bellow).

If *format* is omitted and the expression is numeric, then these functions perform the same function as the **Str$** or **Str** statements, except that they do not preserve a leading space for positive values.

If *expression* is **Null**, then a zero-length string is returned.

The maximum length of the string returned by **Format** or **Format$** functions is 255.

The *firstdayofweek* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofweek*.
ebSunday	1	Sunday (the default)
ebMonday	2	Monday
ebTuesday	3	Tuesday
ebWednesday	4	Wednesday
ebThursday	5	Thursday
ebFriday	6	Friday
ebSaturday	7	Saturday

The *firstdayofyear* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofyear*.
ebFirstJan1	1	The first week of the year is that in which January 1 occurs (the default).

Constant	Value	Description
ebFirstFourDays	2	The first week of the year is that containing at least four days in the year.
ebFirstFullWeek	3	The first week of the year is the first full week of the year.

Built-In Formats

To format numeric expressions, you can specify one of the built-in formats. There are two categories of built-in formats: one deals with numeric expressions and the other with date/time values.The following tables list the built-in numeric and date/time format strings, followed by an explanation of what each does.

Numeric Formats

Format	Description
General Number	Displays the numeric expression as is, with no additional formatting.
Currency	Displays the numeric expression as currency, with thousands separator if necessary. The built-in Currency format allows the specification of an optional user-defined format specification used only for zero values: `Currency;`*zero-format-string* Where *zero-format-string* is a user-defined format used specifically for zero values.
Fixed	Displays at least one digit to the left of the decimal separator and two digits to the right.
Standard	Displays the numeric expression with thousands separator if necessary. Displays at least one digit to the left of the decimal separator and two digits to the right.
Percent	Displays the numeric expression multiplied by 100. A percent sign (%) will appear at the right of the formatted output. Two digits are displayed to the right of the decimal separator.
Scientific	Displays the number using scientific notation. One digit appears before the decimal separator and two after.
Yes/No	Displays No if the numeric expression is 0. Displays Yes for all other values.

Numeric Formats (Continued)

Format	Description
True/False	Displays False if the numeric expression is 0. Displays True for all other values.
On/Off	Displays Off if the numeric expression is 0. Displays On for all other values.

Date/Time Formats

Format	Description
General date	Displays the date and time. If there is no fractional part in the numeric expression, then only the date is displayed. If there is no integral part in the numeric expression, then only the time is displayed. Output is in the following form: 1/1/95 01:00:00 AM.
Medium date	Displays a medium date—prints out only the abbreviated name of the month.
Short date	Displays a short date.
Long time	Displays the long time. The default is: h:mm:ss.
Medium time	Displays the time using a 12-hour clock. Hours and minutes are displayed, and the AM/PM designator is at the end.
Short time	Displays the time using a 24-hour clock. Hours and minutes are displayed.

User-Defined Formats

In addition to the built-in formats, you can specify a user-defined format by using characters that have special meaning when used in a format expression. The following tables list the characters you can use for numeric, string, and date/time formats and explain their functions.

Numeric Formats

Character	Meaning
Empty string	Displays the numeric expression as is, with no additional formatting.

Numeric Formats (Continued)

Character	Meaning
0	This is a digit placeholder. Displays a number or a 0. If a number exists in the numeric expression in the position where the 0 appears, the number will be displayed. Otherwise, a 0 will be displayed. If there are more 0s in the format string than there are digits, the leading and trailing 0s are displayed without modification.
#	This is a digit placeholder. Displays a number or nothing. If a number exists in the numeric expression in the position where the number sign appears, the number will be displayed. Otherwise, nothing will be displayed. Leading and trailing 0s are not displayed.
.	This is the decimal placeholder. Designates the number of digits to the left of the decimal and the number of digits to the right. The character used in the formatted string depends on the decimal placeholder, as specified by your locale.
%	This is the percentage operator. The numeric expression is multiplied by 100, and the percent character is inserted in the same position as it appears in the user-defined format string.
,	This is the thousands separator. The common use for the thousands separator is to separate thousands from hundreds. To specify this use, the thousands separator must be surrounded by digit placeholders. Commas appearing before any digit placeholders are specified are just displayed. Adjacent commas with no digit placeholders specified between them and the decimal mean that the number should be divided by 1,000 for each adjacent comma in the format string. A comma immediately to the left of the decimal has the same function. The actual thousands separator character used depends on the character specified by your locale.

Numeric Formats (Continued)

Character	Meaning
E- E+ e- e+	These are the scientific notation operators, which display the number in scientific notation. At least one digit placeholder must exist to the left of E-, E+, e-, or e+. Any digit placeholders displayed to the left of E-, E+, e-, or e+ determine the number of digits displayed in the exponent. Using E+ or e+ places a + in front of positive exponents and a – in front of negative exponents. Using E- or e- places a – in front of negative exponents and nothing in front of positive exponents.
:	This is the time separator. Separates hours, minutes, and seconds when time values are being formatted. The actual character used depends on the character specified by your locale.
/	This is the date separator. Separates months, days, and years when date values are being formatted. The actual character used depends on the character specified by your locale.
- + $ ( ) space	These are the literal characters you can display. To display any other character, you should precede it with a backslash or enclose it in quotes.
\	This designates the next character as a displayed character. To display characters, precede them with a backslash. To display a backslash, use two backslashes. Double quotation marks can also be used to display characters. Numeric formatting characters, date/time formatting characters, and string formatting characters cannot be displayed without a preceding backslash.
"ABC"	Displays the text between the quotation marks, but not the quotation marks. To designate a double quotation mark within a format string, use two adjacent double quotation marks.
*	This will display the next character as the fill character. Any empty space in a field will be filled with the specified fill character.

Numeric formats can contain one to three parts. Each part is separated by a semicolon. If you specify one format, it applies to all values. If you specify two formats, the first applies to positive values and the second to negative values. If you specify three formats, the first applies to positive values, the second to negative values, and the third to 0s. If you include semicolons with no format between them, the format for positive values is used.

String Formats

Character	Meaning
@	This is a character placeholder. It displays a character if one exists in the expression in the same position; otherwise, it displays a space. Placeholders are filled from right to left unless the format string specifies left to right.
&	This is a character placeholder. It displays a character if one exists in the expression in the same position; otherwise, it displays nothing. Placeholders are filled from right to left unless the format string specifies left to right.
<	This character forces lowercase. It displays all characters in the expression in lowercase.
>	This character forces uppercase. It displays all characters in the expression in uppercase.
!	This character forces placeholders to be filled from left to right. The default is right to left.

Date/Time Formats

Character	Meaning
c	Displays the date as ddddd and the time as ttttt. Only the date is displayed if no fractional part exists in the numeric expression. Only the time is displayed if no integral portion exists in the numeric expression.
d	Displays the day without a leading 0 (1–31).
dd	Displays the day with a leading 0 (01–31).
ddd	Displays the day of the week abbreviated (Sun–Sat).
dddd	Displays the day of the week (Sunday–Saturday).
ddddd	Displays the date as a short date.

Date/Time Formats (Continued)

Character	Meaning
dddddd	Displays the date as a long date.
w	Displays the number of the day of the week (1–7). Sunday is 1; Saturday is 7.
ww	Displays the week of the year (1–53).
m	Displays the month without a leading 0 (1–12). If m immediately follows h or hh, m is treated as minutes (0–59).
mm	Displays the month with a leading 0 (01–12). If mm immediately follows h or hh, mm is treated as minutes with a leading 0 (00–59).
mmm	Displays the month abbreviated (Jan–Dec).
mmmm	Displays the month (January–December).
q	Displays the quarter of the year (1–4).
yy	Displays the year, not the century (00–99).
yyyy	Displays the year (1000–9999).
h	Displays the hour without a leading 0 (0–24).
hh	Displays the hour with a leading 0 (00–24).
n	Displays the minute without a leading 0 (0–59).
nn	Displays the minute with a leading 0 (00–59).
s	Displays the second without a leading 0 (0–59).
ss	Displays the second with a leading 0 (00–59).
ttttt	Displays the time. A leading 0 is displayed if specified by your locale.
AM/PM	Displays the time using a 12-hour clock. Displays an uppercase AM for time values before 12 noon. Displays an uppercase PM for time values after 12 noon and before 12 midnight.

Date/Time Formats (Continued)

Character	Meaning
am/pm	Displays the time using a 12-hour clock. Displays a lowercase am or pm at the end.
A/P	Displays the time using a 12-hour clock. Displays an uppercase A or P at the end.
a/p	Displays the time using a 12-hour clock. Displays a lowercase a or p at the end.
AMPM	Displays the time using a 12-hour clock. Displays the string s1159 for values before 12 noon and s2359 for values after 12 noon and before 12 midnight.

Example

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a# = 1199.234
   message = "Some general formats for '" & a# & "' are:"
   message = message & Format$(a#,"General Number") & crlf
   message = message & Format$(a#,"Currency") & crlf
   message = message & Format$(a#,"Standard") & crlf
   message = message & Format$(a#,"Fixed") & crlf
   message = message & Format$(a#,"Percent") & crlf
   message = message & Format$(a#,"Scientific") & crlf
   message = message & Format$(True,"Yes/No") & crlf
   message = message & Format$(True,"True/False") & crlf
   message = message & Format$(True,"On/Off") & crlf
   message = message & Format$(a#,"0,0.00") & crlf
   message = message & Format$(a#,"##,###,###.###") & crlf
   MsgBox message
   da$ = Date$
   message = "Some date formats for '" & da$ & "' are:"
   message = message & Format$(da$,"General Date") & crlf
   message = message & Format$(da$,"Long Date") & crlf
   message = message & Format$(da$,"Medium Date") & crlf
   message = message & Format$(da$,"Short Date") & crlf
   MsgBox message
   ti$ = Time$
   message = "Some time formats for '" & ti$ & "' are:"
   message = message & Format$(ti$,"Long Time") & crlf
   message = message & Format$(ti$,"Medium Time") & crlf
   message = message & Format$(ti$,"Short Time") & crlf
   MsgBox message
End Sub
```

See Also **Str**, **Str$** (functions); **CStr** (function).

Platform(s) All.

Platform Notes **Windows, Win32:** Under Windows and Win32, default date/time formats are read from the [Intl] section of the win.ini file.

FreeFile (function)

Syntax
```
FreeFile [([rangenumber])]
```

Description Returns an **Integer** containing the next available file number.

Comments This function returns the next available file number within the specified range. If **rangenumber** is 0, then a number between 1 and 255 is returned; if 1, then a number between 256 and 511 is returned. If **rangenumber** is not specified, then a number between 1 and 255 is returned.

The function returns 0 if there is no available file number in the specified range.

The number returned is suitable for use in the **Open** statement.

Example
```
'This example assigns A to the next free file number and
'displays it in a dialog box.
Sub Main()
   a = FreeFile
   MsgBox "The next free file number is: " & a
End Sub
```

See Also **FileAttr** (function); **Open** (statement).

Platform(s) All.

Function...End Function (statement)

Syntax
```
[Private | Public] [Static] Function name[(arglist)] [As ReturnType]
      [statements]
End Sub
```

where *arglist* is a comma-separated list of the following (up to 30 arguments are allowed):
```
[Optional] [ByVal | ByRef] parameter [()] [As type]
```

Description Creates a user-defined function.

Comments The **Function** statement has the following parts:

Part	Description
Private	Indicates that the function being defined cannot be called from other scripts.

Part	Description
Public	Indicates that the function being defined can be called from other scripts. If both the **Private** and **Public** keywords are missing, then **Public** is assumed.
Static	Recognized by the compiler but currently has no effect.
name	Name of the function, which must follow BasicScript naming conventions: 1. Must start with a letter. 2. May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. 3. Must not exceed 80 characters in length. Additionally, the *name* parameter can end with an optional type-declaration character specifying the type of data returned by the function (i.e., any of the following characters: **%**, **&**, **!**, **#**, **@**).
Optional	Keyword indicating that the parameter is optional. All optional parameters must be of type **Variant**. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter is required. **Note:** You can use the **IsMissing** function to determine whether an optional parameter was actually passed by the caller.
ByVal	Keyword indicating that *parameter* is passed by value.
ByRef	Keyword indicating that *parameter* is passed by reference. If neither the **ByVal** nor the **ByRef** keyword is given, then **ByRef** is assumed.
parameter	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of **As** *type*.

type	Type of the parameter (**Integer**, **String**, and so on). Arrays are indicated with parentheses. For example, an array of integers would be declared as follows:
	`Function Test(a() As Integer)`
	`End Function`

Part	Description
ReturnType	Type of data returned by the function. If the return type is not given, then **Variant** is assumed. The *ReturnType* can only be specified if the function name (i.e., the *name* parameter) does not contain an explicit type-declaration character.

A function returns to the caller when either of the following statements is encountered:

```
End Function
Exit Function
```

Functions can be recursive.

Returning Values from Functions

To assign a return value, an expression must be assigned to the name of the function, as shown below:

```
Function TimesTwo(a As Integer) As Integer
   TimesTwo = a * 2
End Function
```

If no assignment is encountered before the function exits, then one of the following values is returned:

Value	Data Type Returned by the Function
0	**Integer**, **Long**, **Single**, **Double**, **Currency**
Zero-length string	**String**
Nothing	**Object** (or any data object)
Error	**Variant**
December 30, 1899	**Date**
False	**Boolean**

The type of the return value is determined by the **As** *ReturnType* clause on the **Function** statement itself. As an alternative, a type-declaration character can be added to the **Function** name. For example, the following two definitions of **Test** both return **String** values:

```
Function Test() As String
   Test = "Hello, world"
End Function
Function Test$()
   Test = "Hello, world"
End Function
```

Functions in BasicScript cannot return user-defined types or dialogs.

Passing Parameters to Functions

Parameters are passed to a function either by value or by reference, depending on the declaration of that parameter in *arglist*. If the parameter is declared using the **ByRef** keyword, then any modifications to that passed parameter within the function change the value of that variable in the caller. If the parameter is declared using the **ByVal** keyword, then the value of that variable cannot be changed in the called function. If neither the **ByRef** or **ByVal** keywords are specified, then the parameter is passed by reference.

You can override passing a parameter by reference by enclosing that parameter within parentheses. For instance, the following example passes the variable j by reference, regardless of how the third parameter is declared in the *arglist* of **UserFunction**:

```
i = UserFunction(10,12,(j))
```

Optional Parameters

BasicScript allows you to skip parameters when calling functions, as shown in the following example:

```
Function Test(a%,b%,c%) As Variant
End Function
Sub Main
   a = Test(1,,4)    'Parameter 2 was skipped.
End Sub
```

You can skip any parameter, with the following restrictions:

1. The call cannot end with a comma. For instance, using the above example, the following is not valid:
   ```
   a = Test(1,,)
   ```
2. The call must contain the minimum number of parameters as required by the called function. For instance, using the above example, the following are invalid:
   ```
   a = Test(,1)      'Only passes two out of three required
                     'parameters.
   a = Test(1,2)     'Only passes two out of three required
                     'parameters.
   ```

When you skip a parameter in this manner, BasicScript creates a temporary variable and passes this variable instead. The value of this temporary variable depends on the data type of the corresponding parameter in the argument list of the called function, as described in the following table:

Value	Data Type
0	**Integer**, **Long**, **Single**, **Double**, **Currency**
Zero-length string	**String**
Nothing	**Object** (or any data object)

Value	Data Type
Error	**Variant**
December 30, 1899	**Date**
False	**Boolean**

Within the called function, you will be unable to determine whether a parameter was skipped unless the parameter was declared as a variant in the argument list of the function. In this case, you can use the **IsMissing** function to determine whether the parameter was skipped:

```
Function Test(a,b,c)
   If IsMissing(a) Or IsMissing(b) Then Exit Sub
End Function
```

Example

```
Function Factorial(n%) As Integer
   'This function calculates N! (N-factoral).
   f% = 1
   For i = n To 2 Step -1
      f = f * i
   Next i
   Factorial = f
End Function
Sub Main()
   'This example calls user-defined function Factoral and
   'displays the result in a dialog box.
   a% = 0
   prompt$ = "Enter an integer number greater than 2."
   Do While a% < 2
      a% = Val(InputBox$(prompt$,"Compute Factorial"))
   Loop
   b# = Factorial(a%)
   MsgBox "The factoral of " & a% & " is: " & b#
End Sub
```

See Also **Sub...End Sub** (statement)

Platform(s) All.

Fv (function)

Syntax `Fv(`*rate*`, `*nper*`, `*pmt*`, `*pv*`, `*due*`)`

Description Calculates the future value of an annuity based on periodic fixed payments and a constant rate of interest.

Comments

An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans.

The **Fv** function requires the following named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period. Make sure that annual rates are normalized for monthly periods (divided by 12).
nper	**Double** representing the total number of payments (periods) in the annuity.
pmt	**Double** representing the amount of each payment per period. Payments are entered as negative values, whereas receipts are entered as positive values.
pv	**Double** representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan, whereas in the case of a retirement annuity, the present value would be the amount of the fund.
due	**Integer** indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 indicates payment at the start of each period.

The *rate* and *nper* values must be expressed in the same units. If *rate* is expressed as a percentage per month, then *nper* must also be expressed in months. If *rate* is an annual rate, then the *nper* value must also be given in years.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example

```
'This example calculates the future value of 100 dollars paid
'periodically for a period of 10 years (120 months) at a rate of
'10% per year (or .10/12 per month) with payments made on the
'first of the month. The value is displayed in a dialog box.
'Note that payments are negative values.
Sub Main()
   a# = Fv((.10/12),120,-100.00,0,1)
   MsgBox "Future value is: " & Format(a#,"Currency")
End Sub
```

See Also

IRR (function); **MIRR** (function); **Npv** (function); **Pv** (function).

Platform(s)

All.

Get (statement)

Syntax `Get [#] filenumber, [recordnumber], variable`

Description Retrieves data from a random or binary file and stores that data into the specified variable.

Comments The **Get** statement accepts the following parameters:

Parameter	Description
filenumber	**Integer** used by BasicScript to identify the file. This is the same number passed to the **Open** statement.
recordnumber	**Long** specifying which record is to be read from the file. For binary files, this number represents the first byte to be read starting with the beginning of the file (the first byte is 1). For random files, this number represents the record number starting with the beginning of the file (the first record is 1). This value ranges from 1 to 2147483647. If the *recordnumber* parameter is omitted, the next record is read from the file (if no records have been read yet, then the first record in the file is read). When this parameter is omitted, the commas must still appear, as in the following example: `Get #1,,recvar` If *recordnumber* is specified, it overrides any previous change in file position specified with the **Seek** statement.
variable	Variable into which data will be read. The type of the variable determines how the data is read from the file, as described below.

With random files, a runtime error will occur if the length of the data being read exceeds the *reclen* parameter specified with the **Open** statement. If the length of the data being read is less than the record length, the file pointer is advanced to the start of the next record. With binary files, the data elements being read are contiguous—the file pointer is never advanced.

Variable Types

The type of the *variable* parameter determines how data will be read from the file. It can be any of the following types:

Variable Type	File Storage Description
Integer	2 bytes are read from the file.
Long	4 bytes are read from the file.

Variable Type	File Storage Description

String (variable-length)	In binary files, variable-length strings are read by first determining the specified string variable's length and then reading that many bytes from the file. For example, to read a string of eight characters:
	`s$=String$(8,"")`
	`Get#1,,s$`
	In random files, variable-length strings are read by first reading a 2-byte length and then reading that many characters from the file.

String (fixed-length)	Fixed-length strings are read by reading a fixed number of characters from the file equal to the string's declared length.
Double	8 bytes are read from the file (IEEE format).
Single	4 bytes are read from the file (IEEE format).
Date	8 bytes are read from the file (IEEE double format).
Boolean	2 bytes are read from the file. Nonzero values are **True**, and zero values are **False**.
Variant	A 2-byte **VarType** is read form the file, which determines the format of the data that follows. Once the **VarType** is known, the data is read individually, as described above. With user-defined errors, after the 2-byte **VarType**, a 2-byte unsigned integer is read and assigned as the value of the user-defined error, followed by 2 additional bytes of information about the error. The exception is with strings, which are always preceded by a 2-byte string length.
User-defined types	Each member of a user-defined data type is read individually. In binary files, variable-length strings within user-defined types are read by first reading a 2-byte length followed by the string's content. This storage is different from variable-length strings outside of user-defined types. When reading user-defined types, the record length must be greater than or equal to the combined size of each element within the data type.
Arrays	Arrays cannot be read from a file using the **Get** statement.

Variable Type	File Storage Description
Object	Object variables cannot be read from a file using the **Get** statement.

Example

```
'This example opens a file for random write, then writes ten
'records into the file with the values 10...50. Then the file is
'closed and reopened in random mode for read, and the records
'are read with the Get statement. The result is displayed in a
'message box.
Sub Main()
   Open "test.dat" For Random Access Write As #1
   For x = 1 to 10
      y% = x * 10
      Put #1,x,y
   Next x
   Close
   Open "test.dat" For Random Access Read As #1
   For y = 1 to 5
      Get #1,y,x%
      message = message & "Record " & y & ": " & x% & Basic.Eoln$
   Next y
   MsgBox message
   Close
End Sub
```

See Also **Open** (statement); **Put** (statement); **Input#** (statement); **Line Input#** (statement); **Input**, **Input$, InputB**, **InputB$** (functions).

Platform(s) All.

GetAllSettings (function)

Syntax `GetAllSettings(`*appname* `[,`*section*`])`

Description Returns all of the keys within the specified section, or all of the sections within the specified application from the system registry.

Comments The **GetAllSettings** function takes the following named parameters:

Named Parameter	Description
appname	A **String** expression specifying the name of the application from which settings or keys will be returned.
section	A **String** expression specifying the name of the section from which keys will be returned. If omitted, then all of the section names within *appname* will be returned.

The **GetAllSettings** function returns a **Variant** containing an array of strings.

Example

```
Sub Main()
   Dim NewAppSettings() As Variant
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Height", setting := 200
   SaveSetting appname := "NewApp", section := "Startup _
      ", key := "Width", setting := 320
   GetAllSettings appname := "NewApp", _
      section := "Startup", resultarray := NewAppSettings
   For i = LBound(NewAppSettings) To UBound(NewAppSettings)
      NewAppSettings(i) = NewAppSettings(i) & "=" & _
         GetSetting("NewApp", "Startup", NewAppSettings(i))
   Next i
   r = SelectBox("Registry Settings","", NewAppSettings)
End Sub
```

See Also **GetSetting** (function); **DeleteSetting** (statement); **SaveSetting** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Win32:** Under Win32, this statement operates on the system registry. All settings are read from the following entry in the system registry:

HKEY_CURRENT_USER\Software\BasicScript Program Settings*appname**section*

Windows, OS/2: Settings are stored in INI files. The name of the INI file is specified by *appname*. If *appname* is omitted, then this command operates on the WIN.INI file. For example, to enumerate all of the keys within the **intl** section of the WIN.INI file, you could use the following statements:

```
Dim a As Variant
a = GetAllSettings(,"intl")
```

GetAttr (function)

Syntax `GetAttr(`*pathname*`)`

Description Returns an **Integer** containing the attributes of the specified file.

Comments The attribute value returned is the sum of the attributes set for the file. The value of each attribute is as follows:

Constant	Value	Includes
ebNormal	0	Read-only files, archive files, subdirectories, and files with no attributes
ebReadOnly	1	Read-only files
ebHidden	2	Hidden files

Constant	Value	Includes
ebSystem	4	System files
ebVolume	9	Volume label
ebDirectory	16	Subdirectories
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

To deterimine whether a particular attribute is set, you can **And** the values shown above with the value returned by **GetAttr**. If the result is **True**, the attribute is set, as shown below:

```
Dim w As Integer
w = GetAttr("sample.txt")
If w And ebReadOnly Then MsgBox "This file is read-only."
```

Example

```
'This example tests to see whether the file test.dat exists. If
'it does not, then it creates the file. The file attributes are
'then retrieved with the GetAttr function, and the result is
'displayed.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   If Not FileExists("test.dat") Then
      Open "test.dat" For Random Access Write As #1
      Close
   End If
   y% = GetAttr("test.dat")
   If y% And ebNone Then _
      message = message & "No archive bit is set." & crlf
   If y% And ebReadOnly Then _
      message = message & "The read-only bit is set." & crlf
   If y% And ebHidden Then _
      message = message & "The hidden bit is set." & crlf
   If y% And ebSystem Then _
      message = message & "The system bit is set." & crlf
   If y% And ebVolume Then _
      message = message & "The volume bit is set." & crlf
   If y% And ebDirectory Then _
      message = message & "The directory bit is set." & crlf
   If y% And ebArchive Then _
      message = message & "The archive bit is set."
   MsgBox message
   Kill "test.dat"
End Sub
```

See Also **SetAttr** (statement); **FileAttr** (function).

Platform(s) All.

Platform Notes **Windows:** Under Windows, these attributes are the same as those used by DOS.

UNIX: On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

GetCheckBox (function)

Syntax `GetCheckBox(`*name$* `|` *id*`)`

Description Returns an **Integer** representing the state of the specified check box.

Comments This function is used to determine the state of a check box, given its name or ID. The returned value will be one of the following:

Returned Value	Description
0	Check box contains no check.
1	Check box contains a check.
2	Check box is grayed.

The **GetCheckBox** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the check box.
id	**Integer** specifying the ID of the check box.

Note: The **GetCheckBox** function is used to retrieve the state of a check box in another application's dialog box. Use the **DlgValue** function to retrieve the state of a check box in a dynamic dialog box.

Example

```
'This example toggles the Match Case check box in the Find
'dialog box.
Sub Main()
   Menu "Search.Find"
   If GetCheckBox("Match Case") = 0 Then
      SetCheckBox "Match Case",1
   Else
      SetCheckBox "Match Case",0
   End If
End Sub
```

See Also **CheckBoxExists** (function); **CheckBoxEnabled** (function); **SetCheckBox** (statement); **DlgValue** (function).

Platform(s) Windows.

GetComboBoxItem$ (function)

Syntax `GetComboBoxItem$(`*name$* | *id* `[,`*ItemNumber*`])`

Description Returns a **String** containing the text of an item within a combo box.

Comments The **GetComboBoxItem$** function takes the following parameters:

Parameter	Description
name$	**String** specifying the name of the combo box containing the item to be returned. The name of a combo box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a combo box. A runtime error is generated if a combo box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the combo box containing the item to be returned.
ItemNumber	**Integer** containing the line number of the desired combo box item to be returned. If omitted, then the currently selected item in the combo box is returned.

The combo box must exist within the current window or dialog box; otherwise, a runtime error is generated.

A zero-length string will be returned if the combo box does not contain textual items.

Note: The **GetComboBoxItem$** function is used to retrieve the current item of a combo box in another application's dialog box. Use the **DlgText** function to retrieve the current item of a combo box in a dynamic dialog box.

Example

```
'This example retrieves the last item from a combo box.
Sub Main()
   last% = GetComboBoxItemCount("Directories:")
   s$ = GetComboBoxItem$("Directories:",last% - 1) 'Number is
                                                   '0-based.
   MsgBox "The last item in the combo box is " & s$
End Sub
```

See Also **ComboBoxEnabled** (function); **ComboBoxExists** (function); **GetComboBoxItemCount** (function); **SelectComboBoxItem** (statement).

Platform(s) Windows.

GetComboBoxItemCount (function)

Syntax `GetComboBoxItemCount(`*name$* | *id*`)`

Description Returns an **Integer** containing the number of items in the specified combo box.

Comments The **GetComboBoxItemCount** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the combo box. The name of a combo box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a combo box. A runtime error is generated if a combo box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the combo box.

A runtime error is generated if the specified combo box does not exist within the current window or dialog box.

Note: The **GetComboBoxItemCount** function is used to determine the number of items in a combo box in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This example copies all the items out of a combo box and into
'an array.
Sub Main()
   Dim MyList$()
   last% = GetComboBoxItemCount("Directories:")
   ReDim MyList$(0 To last - 1)
   For i = 0 To last - 1
      MyList$(i) = GetComboBoxItem$("Directories:",i)
   Next i
End Sub
```

See Also **ComboBoxEnabled** (function); **ComboBoxExists** (function); **GetComboBoxItem$** (function); **SelectComboBoxItem** (statement).

Platform(s) Windows.

GetEditText$ (function)

Syntax `GetEditText$(`*name$* | *id*`)`

Description Returns a **String** containing the content of the specified text box control.

Comments The **GetEditText$** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the text box whose content will be returned. The name of a text box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a text box. A runtime error is generated if a text box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the text box whose content will be returned.

A runtime error is generated if a text box control with the given name or ID cannot be found within the active window.

Note: The **GetEditText$** function is used to retrieve the content of a text box in another application's dialog box. Use the **DlgText$** function to retrieve the content of a text box in a dynamic dialog box.

Example

```
'This example retrieves the filename and prepends it with the
'current directory.
Sub Main()
   s$ = GetEditText$("Filename:")'Retrieve edit control content.
   s$ = CurDir$ & Basic.PathSeparator & s$'Prepend current dir.
   SetEditText "Filename:",s$'Put it back.
End Sub
```

See Also **EditEnabled** (function); **EditExists** (function); **SetEditText** (statement).

Platform(s) Windows.

GetListBoxItem$ (function)

Syntax `GetListBoxItem$(`*name$* | *id*`,[`*item*`])`

Description Returns a String containing the specified item in a list box.

Comments The **GetListBoxItem$** function takes the following parameters:

Parameter	Description
name$	**String** specifying the name of the list box containing the item to be returned.

Parameter	Description
	The name of a list box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a list box. A runtime error is generated if a list box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the list box containing the item to be returned.
item	**Integer** containing the line number of the desired list box item to be returned. This number must be between 1 and the number of items in the list box. If omitted, then the currently selected item in the list box is returned.

A runtime error is generated if the specified list box cannot be found within the active window.

Note: The **GetListBoxItem$** function is used to retrieve an item from a list box in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This example sees whether my name appears as an item in the
'"Users" list box.
Sub Main()
   last% = GetListBoxItemCount("Users")
   IsThere = False
   For i = 0 To last% - 1'Number is zero-based.
      If GetListBoxItem$("Users",i) = Net.User$ Then _
         isThere = True
   Next i
   If IsThere Then MsgBox "I am a member!",ebOKOnly
End Sub
```

See Also **GetListBoxItemCount** (function); **ListBoxEnabled** (function); **ListBoxExists** (function); **SelectListBoxItem** (statement).

Platform(s) Windows.

GetListBoxItemCount (function)

Syntax `GetListBoxItemCount(`*name$* | *id*`)`

Description Returns an **Integer** containing the number of items in a specified list box.

Comments The **GetListBoxItemCount** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the list box. The name of a list box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a list box. A runtime error is generated if a list box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the list box.

A runtime error is generated if the specified list box cannot be found within the active window.

Note: The **GetListBoxItemCount** function is used to retrieve the number of items in a list box in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example See **GetListBoxItem$** (function).

See Also **GetListBoxItem$** (function); **ListBoxEnabled** (function); **ListBoxExists** (function); **SelectListBoxItem** (statement).

Platform(s) Windows.

GetObject (function)

Syntax `GetObject(`*pathname* `[,` *class*`])`

Description Returns the object specified by *pathname* or returns a previously instantiated object of the given *class*.

Comments This function is used to retrieve an existing OLE Automation object, either one that comes from a file or one that has previously been instantiated.

The *pathname* argument specifies the full pathname of the file containing the object to be activated. The application associated with the file is determined by OLE at runtime. For example, suppose that a file called c:\docs\resume.doc was created by a word processor called wordproc.exe. The following statement would invoke wordproc.exe, load the file called c:\docs\resume.doc, and assign that object to a variable:

```
Dim doc As Object
Set doc = GetObject("c:\docs\resume.doc")
```

To activate a part of an object, add an exclamation point to the filename followed by a string representing the part of the object that you want to activate. For example, to activate the first three pages of the document in the previous example:

```
Dim doc As Object
Set doc = GetObject("c:\docs\resume.doc!P1-P3")
```

The **GetObject** function behaves differently depending on whether the first named parameter is omitted. The following table summarizes the different behaviors of **GetObject**:

pathname	*class*	**GetObject Returns**
Not specified	Specified	A reference to an existing instance of the specified object. A runtime error results if the object is not already loaded.
" "	Specified	A reference to a new object (as specified by *class*). A runtime error occurs if an object of the specified class cannot be found. This is the same as **CreateObject**.
Specified	Not specified	The default object from *pathname*. The application to activate is determined by OLE based on the given filename.
Specified	Specified	The object given *class* from the file given by *pathname*. A runtime error occurs if an object of the given class cannot be found in the given file.

Examples

```
'This first example instantiates the existing copy of Excel.
Dim Excel As Object
Set Excel = GetObject(,"Excel.Application")
'This second example loads the OLE server associated with a
'document.
Dim MyObject As Object
Set MyObject = GetObject("c:\documents\resume.doc",)
```

See Also **CreateObject** (function); **Object** (data type).

Platform(s) Windows, Win32, Macintosh.

GetOption (function)

Syntax `GetOption(`*name$* `|` *id*`)`

Description Returns **True** if the option is set; returns **False** otherwise.

Comments The **GetOption** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the option button.

Parameter	Description
id	**Integer** containing the ID of the option button. The *id* must be used when the name of the option button is not known in advance.

The option button must exist within the current window or dialog box.

A runtime error will be generated if the specified option button does not exist.

Note: The **GetOption** function is used to retrieve the state of an option button in another application's dialog box. Use the **DlgValue** function to retrieve the state of an option button in a dynamic dialog box.

Example

```
'This example figures out which option is set in the Desktop
'dialog box of the Control Panel.
Sub Main()
   id = Shell("control",7)'Run the Control Panel.
   WinActivate "Control Panel"'Activate the Control Panel window.
   Menu "Settings.Desktop"'Select Desktop dialog box.
   WinActivate "Control Panel|Desktop"'Activate it.
   If GetOption("Tile") Then'Retrieve which option is set.
      MsgBox "Your wallpaper is tiled."
   Else
      MsgBox "Your wallpaper is centered."
   End If
End Sub
```

See Also **OptionEnabled** (function); **OptionExists** (function); **SetOption** (statement).

Platform(s) Windows.

GetSetting (function)

Syntax `GetSetting([`*appname*`], `*section*`, `*key*`[, `*default*`])`

Description Retrieves an specific setting from the system registry.

Comments The **GetSetting** function has the following named parameters:

Named Parameter	Description
appname	A **String** expression specifying the name of the application from which the setting will be read.
section	A **String** expression specifying the name of the section within *appname* to be read.

Named Parameter	Description
key	A **String** expression specifying the name of the key within *section* to be read.
default	An optional **String** expression specifying the default value to be returned if the desired key does not exist in the system registry. If omitted, then an empty string is returned if the key doesn't exist.

Example

```
Sub Main()
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Height", setting := 200
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Width", setting := 320
   MsgBox GetSetting(appname := "NewApp", section := "Startup", _
      key := "Height", default := "50")
   DeleteSetting "NewApp"' Delete the NewApp key
End Sub
```

See Also **GetAllSettings** (function); **DeleteSetting** (statement); **SaveSetting** (statement).

Platform(s) Win32, Windows, OS/2.

Platform Notes **Win32:** Under Win32, this statement operates on the system registry. All settings are read from the following entry in the system registry:

```
HKEY_CURRENT_USER\Software\BasicScript Program
Settings\appname\section\key
```

On this platform, the *appname* parameter is not optional.

Windows, OS/2: Settings are stored in INI files. The name of the INI file is specified by *appname*. If *appname* is omitted, then this command operates on the WIN.INI file. For example, to read the **sLanguage** setting from the **intl** section of the WIN.INI file, you could use the following statement:

```
s$ = GetSetting(,"intl","sLanguage")
```

Global (statement)

Description See **Public** (statement).

Platform(s) All.

GoSub (statement)

Syntax `GoSub` *label*

Description Causes execution to continue at the specified label.

Comments Execution can later be returned to the statement following the **GoSub** by using the **Return** statement.

The *label* parameter must be a label within the current function or subroutine. **GoSub** outside the context of the current function or subroutine is not allowed.

Example

```
'This example gets a name from the user and then branches to a
'subroutine to check the input. If the user clicks Cancel or
'enters a blank name, the program terminates; otherwise, the
'name is set to MICHAEL, and a message is displayed.
Sub Main()
   uname$ = Ucase$(InputBox$("Enter your name:","Enter Name"))
   GoSub CheckName
   MsgBox "Hello, " & uname$
   Exit Sub
CheckName:
   If (uname$ = "") Then
      GoSub BlankName
   ElseIf uname$ = "MICHAEL" Then
      GoSub RightName
   Else
      GoSub OtherName
   End If
   Return
BlankName:
   MsgBox "No name? Clicked Cancel? I'm shutting down."
   Exit Sub
RightName:
   Return
OtherName:
   MsgBox "I am renaming you MICHAEL!"
   uname$ = "MICHAEL"
   Return
End Sub
```

See Also **Goto** (statement); **Return** (statement).

Platform(s) All.

Goto (statement)

Syntax `Goto` *label*

Description Transfers execution to the line containing the specified label.

Comments The compiler will produce an error if *label* does not exist.

The *label* must appear within the same subroutine or function as the **Goto**.

Labels are identifiers that follow these rules:

1. Must begin with a letter.
2. May contain letters, digits, and the underscore character.
3. Must not exceed 80 characters in length.
4. Must be followed by a colon (:).

Labels are not case-sensitive.

Example

```
'This example gets a name from the user and then branches to a
'statement, depending on the input name. If the name is not
'MICHAEL, it is reset to MICHAEL unless it is null or the user
'clicks Cancel--in which case, the program displays a message
'and terminates.
Sub Main()
   uname$ = Ucase$(InputBox$("Enter your name:","Enter Name"))
   If uname$ = "MICHAEL" Then
      Goto RightName
   Else
      Goto WrongName
   End If
WrongName:
   If (uname$ = "") Then
      MsgBox "No name? Clicked Cancel? I'm shutting down."
   Else
      MsgBox "I am renaming you MICHAEL!"
      uname$ = "MICHAEL"
      Goto RightName
   End If
   Exit Sub
RightName:
   MsgBox "Hello, MICHAEL!"
End Sub
```

See Also **GoSub** (statement); **Call** (statement).

Platform(s) All.

Platform Notes **Windows, Win32:** To break out of an infinite loop, press Ctrl+Break.

UNIX: To break out of an infinite loop, press Ctrl+C.

Macintosh: To break out of an infinite loop, press Ctrl+Period.

OS/2: To break out of an infinite loop, press Ctrl+C or Ctrl+Break.

GroupBox (statement)

Syntax `GroupBox` *x*,*y*,*width*,*height*,*title$* [,*.Identifier*]

Description Defines a group box within a dialog box template.

Comments This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The group box control is used for static display only—the user cannot interact with a group box control.

Separator lines can be created using group box controls. This is accomplished by creating a group box that is wider than the width of the dialog box and extends below the bottom of the dialog box—i.e., three sides of the group box are not visible.

If *title$* is a zero-length string, then the group box is drawn as a solid rectangle with no title.

The **GroupBox** statement requires the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** coordinates specifying the dimensions of the control in dialog units.
title$	**String** containing the label of the group box. If *title$* is a zero-length string, then no title will appear.
.Identifier	Optional parameter that specifies the name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). If omitted, then the first two words of *title$* are used.

Example

```
'This example shows the GroupBox statement being used both for
'grouping and as a separator line.
Sub Main()
   Begin Dialog OptionsTemplate 16,32,128,84,"Options"
      GroupBox 4,4,116,40,"Window Options"
      CheckBox 12,16,60,8,"Show &Toolbar",.ShowToolbar
      CheckBox 12,28,68,8,"Show &Status Bar",.ShowStatusBar
      GroupBox -12,52,152,48," ",.SeparatorLine
      OKButton 16,64,40,14,.OK
      CancelButton 68,64,40,14,.Cancel
   End Dialog
   Dim OptionsDialog As OptionsTemplate
   Dialog OptionsDialog
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, OS/2, Macintosh, UNIX.

HelpButton (statement)

Syntax `HelpButton` *x*,*y*,*width*,*height*,*HelpFileName$*,*HelpContext*, [,*.Identifier*]

Description Defines a help button within a dialog template.

Comments This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **HelpButton** statement takes the following parameters:

Parameter	Description
x,*y*	**Integer** position of the control (in dialog units) relative to the upper left corner of the dialog box.
width,*height*	**Integer** dimensions of the control in dialog units.
HelpFileName$	**String** expression specifying the name of the help file to be invoked when the button is selected.
HelpContext	**Long** expression specifying the ID of the topic within *HelpFileName$* containing context-sensitive help.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

When the user selects a help button, the associated help file is located at the indicated topic. Selecting a help button does not remove the dialog. Similarly, no actions are sent to the dialog procedure when a help button is selected.

When a help button is present within a dialog, it can be automatically selected by pressing the help key (F1 on most platforms).

Example

```
Sub Main()
   Begin Dialog HelpDialogTemplate ,,180,96,"Untitled"
      OKButton 132,8,40,14
      CancelButton 132,28,40,14
      HelpButton 132,48,40,14,"", 10
      Text 16,12,88,12,"Please click ""Help""",.Text1
   End Dialog
```

```
Dim HelpDialog As
HelpDialogTemplat
e

Dialog HelpDialog
```

```
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **Begin Dialog** (statement); **PictureButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Hex, Hex$ (functions)

Syntax `Hex[$](`*number*`)`

Description Returns a **String** containing the hexadecimal equivalent of *number*.

Comments **Hex$** returns a **String**, whereas **Hex** returns a **String** variant.

The returned string contains only the number of hexadecimal digits necessary to represent the number, up to a maximum of eight.

The *number* parameter can be any type but is rounded to the nearest whole number before converting to hex. If the passed number is an integer, then a maximum of four digits are returned; otherwise, up to eight digits can be returned.

The *number* parameter can be any expression convertible to a number. If *number* is **Null**, then **Null** is returned. **Empty** is treated as 0.

Example

```
'This example inputs a number and displays it in decimal and
'hex until the input number is 0 or an invalid input.
Sub Main()
   Do
      xs$ = InputBox$("Enter a number to convert:","Hex Convert")
      x = Val(xs$)
      If x <> 0 Then
         MsgBox "Dec: " & x & "  Hex: " & Hex$(x)
      Else
         MsgBox "Goodbye."
      End If
   Loop While x <> 0
End Sub
```

See Also **Oct**, **Oct$** (functions).

Platform(s) All.

HLine (statement)

Syntax `HLine [`*lines*`]`

Description Scrolls the window with the focus left or right by the specified number of lines.

Comments The *lines* parameter is an **Integer** specifying the number of lines to scroll. If this parameter is omitted, then the window is scrolled right by one line.

Example

```
'This example scrolls the Notepad window to the left by three
'"amounts." Each "amount" is equivalent to clicking the right
'arrow of the horizontal scroll bar once.
Sub Main()
   AppActivate "Notepad"
   HLine 3     'Move 3 lines in.
End Sub
```

See Also **HPage** (statement); **HScroll** (statement).

Platform(s) Windows, Win32.

Hour (function)

Syntax `Hour(`*time*`)`

Description Returns the hour of the day encoded in the specified *time* parameter.

Comments The value returned is as an **Integer** between 0 and 23 inclusive.

The *time* parameter is any expression that converts to a **Date**.

Example

```
'This example takes the current time; extracts the hour, minute,
'and second; and displays them as the current time.
Sub Main()
   xt# = TimeValue(Time$())
   xh# = Hour(xt#)
   xm# = Minute(xt#)
   xs# = Second(xt#)
   MsgBox "The current time is: " & xh# & ":" & xm# & ":" & xs#
End Sub
```

See Also **Day** (function); **Minute** (function); **Second** (function); **Month** (function); **Year** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

HPage (statement)

Syntax `HPage [`*pages*`]`

Description Scrolls the window with the focus left or right by the specified number of pages.

Comments The *pages* parameter is an **Integer** specifying the number of pages to scroll. If this parameter is omitted, then the window is scrolled right by one page.

Example

```
'This example scrolls the Notepad window to the left by three
'"amounts." Each "amount" is equivalent to clicking within the
'horizontal scroll bar on the right side of the thumb mark.
Sub Main()
   AppActivate "Notepad"
   HPage 3  'Move 3 pages down.
End Sub
```

See Also **HLine** (statement); **HScroll** (statement).

Platform(s) Windows, Win32.

HScroll (statement)

Syntax `HScroll` *percentage*

Description Sets the thumb mark on the horizontal scroll bar attached to the current window.

Comments The position is given as a percentage of the total range associated with that scroll bar. For example, if the *percentage* parameter is 50, then the thumb mark is positioned in the middle of the scroll bar.

Example

```
'This example centers the thumb mark on the horizontal scroll
'bar of the Notepad window.
Sub Main()
   AppActivate "Notepad"
   HScroll 50     'Jump to the middle of the document.
End Sub
```

See Also **HLine** (statement); **HPage** (statement).

Platform(s) Windows, Win32.

HWND (object)

Syntax `Dim` *name* `As HWND`

Description A data type used to hold window objects.

Comments This data type is used to hold references to physical windows in the operating environment. The following commands operate on **HWND** objects:

```
WinActivate    WinClose       WinFind        WinList

WinMaximize    WinMinimize    WinMove        WinRestore

WinSize
```

The above language elements support both string and **HWND** window specifications.

Example

```
'This example activates the "Main" MDI window within Program
'Manager.
Sub Main()
   Dim ProgramManager As HWND
   Dim ProgramManagerMain As HWND
   Set ProgramManager = WinFind("Program Manager")
   If ProgramManager Is Not Nothing Then
      WinActivate ProgramManager
      WinMaximize ProgramManager
      Set ProgramManagerMain = WinFind("Program Manager|Main")
      If ProgramManagerMain Is Not Nothing Then
         WinActivate ProgramManagerMain
         WinRestore ProgramManagerMain
      Else
         MsgBox "Your Program Manager doesn't have a Main group."
      End If
   Else
      MsgBox "Program Manager is not running."
   End If
End Sub
```

See Also **HWND.Value** (property); **WinFind** (function); **WinActivate** (statement).

Platform(s) Windows, Win32.

HWND.Value (property)

Syntax *window*.Value

Description The default property of an **HWND** object that returns a **Variant** containing a **HANDLE** to the physical window of an **HWND** object variable.

Comments The **Value** property is used to retrieve the operating environment–specific value of a given **HWND** object. The size of this value depends on the operating environment in which the script is executing and thus should always be placed into a **Variant** variable.

This property is read-only.

Example

```
'This example displays a dialog box containing the class name of
'Program Manager's Main window. It does so using the .Value
'property, passing it directly to a Windows external routine.
Declare Sub GetClassName Lib "user" (ByVal Win%,ByVal
ClsName$,ByVal ClsNameLen%)
Sub Main()
   Dim ProgramManager As HWND
   Set ProgramManager = WinFind("Program Manager")
   ClassName$ = Space(40)
   GetClassName ProgramManager.Value,ClassName$,Len(ClassName$)
   MsgBox "The program classname is: " & ClassName$
End Sub
```

See Also **HWND** (object).

Platform(s) Windows, Win32.

Platform Notes Under Windows, this value is an **Integer**. Under Win32, this value is a **Long**.

If...Then...Else (statement)

Syntax 1

```
If condition Then statements [Else else_statements]
```

Syntax 2

```
If condition Then
   [statements]
[ElseIf else_condition Then
   [elseif_statements]]
[Else
   [else_statements]]
End If
```

Description Conditionally executes a statement or group of statements.

Comments The single-line conditional statement (syntax 1) has the following parameters:

Parameter	Description
condition	Any expression evaluating to a **Boolean** value.
statements	One or more statements separated with colons. This group of statements is executed when *condition* is **True**.
else_statements	One or more statements separated with colons. This group of statements is executed when *condition* is **False**.

The multiline conditional statement (syntax 2) has the following parameters:

Parameter	Description
condition	Any expression evaluating to a **Boolean** value.

Parameter	Description
statements	One or more statements to be executed when *condition* is **True**.
else_condition	Any expression evaluating to a **Boolean** value. The *else_condition* is evaluated if *condition* is **False**.
elseif_statements	One or more statements to be executed when *condition* is **False** and *else_condition* is **True**.
else_statments	One or more statements to be executed when both *condition* and *else_condition* are **False**.

There can be as many **ElseIf** conditions as required.

Example

```
'This example inputs a name from the user and checks to see
'whether it is MICHAEL or MIKE using three forms of the
'If...Then...Else statement. It then branches to a statement
'that displays a welcome message depending on the user's name.
Sub Main()
   uname$ = UCase$(InputBox$("Enter your name:","Enter Name"))
   If uname$ = "MICHAEL" Then GoSub MikeName
   If uname$ = "MIKE" Then
      GoSub MikeName
      Exit Sub
   End If
   If uname$ = "" Then
      MsgBox "Since you don't have a name, I'll call you MIKE!"
      uname$ = "MIKE"
      GoSub MikeName
   ElseIf uname$ = "MICHAEL" Then
      GoSub MikeName
   Else
      GoSub OtherName
   End If
   Exit Sub
MikeName:
   MsgBox "Hello, MICHAEL!"
   Return
OtherName:
   MsgBox "Hello, " & uname$ & "!"
   Return
End Sub
```

See Also **Choose** (function); **Switch** (function); **IIf** (function); **Select...Case** (statement).

Platform(s) All.

IIf (function)

Syntax `IIf(`*expression*`, `*truepart*`, `*falsepart*`)`

Description Returns *truepart* if condition is **True**; otherwise, returns *falsepart*.

Comments Both expressions are calculated before **IIf** returns.

The **IIf** function is shorthand for the following construct:

```
If condition Then
   variable = truepart
Else
   variable = falsepart
End If
```

Example

```
Sub Main()
   s$ = "Car"
   MsgBox IIf(s$ = "Car","Nice Car","Nice Automobile")
End Sub
```

See Also **Choose** (function); **Switch** (function); **If...Then...Else** (statement); **Select...Case** (statement).

Platform(s) All.

IMEStatus (function)

Syntax `IMEStatus[()]`

Description Returns the current status of the input method editor.

Comments The **IMEStatus** function returns one of the following constants for Japanese locales:

Constant	Value	Description
ebIMENoOp	0	IME not installed.
ebIMEOn	1	IME on.
ebIMEOff	2	IME off.
ebIMEDisabled	3	IME disabled.
ebIMEHiragana	4	Hiragana double-byte character.
ebIMEKatakanaDbl	5	Katakana double-byte characters.
ebIMEKatakanaSng	6	Katakana single-byte characters.
ebIMEAlphaDbl	7	Alphanumeric double-byte characters.
ebIMEAlphaSng	8	Alphanumeric single-byte characters.

For Chinese locales, one of the following constants are returned:

Constant	Value	Description
ebIMENoOp	0	IME not installed.
ebIMEOn	1	IME on.
ebIMEOff	2	IME off.

For Korean locales, this function returns a value with the first 5 bits having the following meaning:

Bit	If not set (or 0)	If set (or 1)
Bit 0	IME not installed	IME installed
Bit 1	IME disabled	IME enabled
Bit 2	English mode	Hangeul mode
Bit 3	Banja mode (single-byte)	Junja mode (double-byte)
Bit 4	Normal mode	Hanja conversion mode

Note: You can test for the different bits using the **And** operator as follows:

```
a = IMEStatus()
If a And 1 Then ...        'Test for bit 0
If a And 2 Then ...        'Test for bit 1
If a And 4 Then ...        'Test for bit 2
If a And 8 Then ...        'Test for bit 3
If a And 16 Then ...       'Test for bit 4
```

This function always returns 0 if no input method editor is installed.

Example

```
'This example retrieves the IMEStatus and displays the results.
Sub Main()
   a = IMEStatus()
   Select case a
      Case 0
         MsgBox "IME not installed."
      Case 1
         MsgBox "IME on."
      Case 2
         Msgbox "IME off."
   End Select
End Sub
```

See Also Constants (topic).

Platform(s) Windows, Win32, OS/2, Macintosh, UNIX.

Imp (operator)

Syntax *result* = *expression1* `Imp` *expression2*

Description Performs a logical or binary implication on two expressions.

Comments If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical implication is performed as follows:

If *expression1* **is**	**and** *expression2* **is**	**then the** *result* **is**
True	**True**	**True**
True	**False**	**False**
True	**Null**	**Null**
False	**True**	**True**
False	**False**	**True**
False	**Null**	**True**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

Binary Implication

If the two expressions are **Integer**, then a binary implication is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long** and a binary implication is then performed, returning a **Long** result.

Binary implication forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions, according to the following table:

If bit in *expression1* **is**	**and bit in** *expression2* **is**	**the** *result* **is**
1	1	1
0	1	1
1	0	0
0	0	1

Example

```
'This example compares the result of two expressions to
'determine whether one implies the other.
Sub Main()
   a = 10 : b = 20 : c = 30 : d = 40
   If (a < b) Imp (c < d) Then
      MsgBox "a less than b implies that c is less than d."
   Else
      MsgBox "a less than b does not imply that c is less than d."
```

```
   End If
   If (a < b) Imp (c > d) Then
      MsgBox "a less than b implies that c is greater than d."
   Else
      MsgBox "a less than b does not imply that c greater than d."
   End If
End Sub
```

See Also Operator Precedence (topic); **Or** (operator); **Xor** (operator); **Eqv** (operator); **And** (operator).

Platform(s) All.

Inline (statement)

Syntax

```
Inline name [parameters]
   anytext
End Inline
```

Description Allows execution or interpretation of a block of text.

Comments The **Inline** statement takes the following parameters:

Parameter	Description
name	Identifier specifying the type of inline statement
parameters	Comma-separated list of parameters.
anytext	Text to be executed by the **Inline** statement. This text must be in a format appropriate for execution by the **Inline** statement. The end of the text is assumed to be the first occurrence of the words **End Inline** appearing on a line.

Example

```
Sub Main()
   Inline MacScript
      -- AppleScript comment.
      Beep
      Display Dialog "AppleScript" buttons "OK"
   End Inline
End Sub
```

See Also **MacScript** (statement).

Platform(s) All.

Input# (statement)

Syntax `Input [#]`*filenumber%*`,`*variable*`[,`*variable*`]...`

Description Reads data from the file referenced by *filenumber* into the given variables.

Comments Each *variable* must be type-matched to the data in the file. For example, a **String** variable must be matched to a string in the file.

The following parsing rules are observed while reading each variable in the variable list:

1. Leading white space is ignored (spaces and tabs).
2. When reading **String** variables, if the first character on the line is a quotation mark, then characters are read up to the next quotation mark or the end of the line, whichever comes first. Blank lines are read as empty strings. If the first character read is not a quotation mark, then characters are read up to the first comma or the end of the line, whichever comes first. String delimiters (quotes, comma, end-of-line) are not included in the returned string. Spaces are trimmed from the end of unquoted strings.
3. When reading numeric variables, scanning of the number stops when the first non-numeric character (such as a comma, a letter, or any other unexpected character) is encountered. Numeric errors are ignored while reading numbers from a file. The resultant number is automatically converted to the same type as the variable into which the value will be placed. If there is an error in conversion, then 0 is stored into the variable.

   After reading the number, input is skipped up to the next delimiter—a comma, an end-of-line, or an end-of-file.

   Numbers must adhere to any of the following syntaxes:

   `[-|+]`*digits*`[.`*digits*`][E[-|+]`*digits*`][!|#|%|&|@]`
   `&H`*hexdigits*`[!|#|%|&]`
   `&[O]`*octaldigits*`[!|#|%|&|@]`
4. When reading *Boolean* variables, the first character must be #; otherwise, a runtime error occurs. If the first character is #, then input is scanned up to the next delimiter (a comma, an end-of-line, or an end-of-file). If the input matches #FALSE#, then **False** is stored in the **Boolean**; otherwise, **True** is stored.
5. When reading **Date** variables, the first character must be #; otherwise, a runtime error occurs. If the first character is #, then the input is scanned up to the next delimiter (a comma, an end-of-line, or an end-of-file). If the input ends in a # and the text between the #'s can be correctly interpreted as a date, then the date is stored; otherwise, December 31, 1899, is stored.

   Normally, dates that follow the universal date format are input from sequential files. These dates use this syntax:

#YYYY-MM-DD HH:MM:SS#

where *YYYY* is a year between 100 and 9999, *MM* is a month between 1 and 12, *DD* is a day between 1 and 31, *HH* is an hour between 0 and 23, *MM* is a minute between 0 and 59, and *SS* is a second between 0 and 59.

6. When reading **Variant** variables, if the data begins with a quotation mark, then a string is read consisting of the characters between the opening quotation mark and the closing quotation mark, end-of-line, or end-of-file.

   If the input does not begin with a quotation mark, then input is scanned up to the next comma, end-of-line, or end-of-file and a determination is made as to what data is being represented. If the data cannot be represented as a number, **Date**, **Error**, **Boolean**, or **Null**, then it is read as a string.

   The following table describes how special data is interpreted as variants:

Blank line	Read as an **Empty** variant.
#**NULL**#	Read as a **Null** variant.
TRUE#	Read as a **Boolean** variant.
#**FALSE**#	Read as a **Boolean** variant.
ERROR *code*#	Read as a user-defined error.
date#	Read as a **Date** variant.
"*text*"	Read as a **String** variant.

7. If an error occurs in interpretation of the data as a particular type, then that data is read as a **String** variant.
8. When reading numbers into variants, the optional type-declaration character determines the **VarType** of the resulting variant. If no type-declaration character is specified, then BasicScript will read the number according to the following rules:

- **Rule 1:** If the number contains a decimal point or an exponent, then the number is read as **Currency**. If there is an error converting to **Currency**, then the number is treated as a **Double**.
- **Rule 2:** If the number does not contain a decimal point or an exponent, then the number is stored in the smallest of the following data types that most accurately represents that value: **Integer**, **Long**, **Currency**, **Double**.

9. End-of-line is interpreted as either a single line feed, a single carriage return, or a carriage-return/line-feed pair. Thus, text files from any platform can be interpreted using this command.

   The *filenumber* parameter is a number that is used by BasicScript to refer to the open file—the number passed to the **Open** statement.

The *filenumber* must reference a file opened in **Input** mode. It is good practice to use the **Write** statement to write date elements to files read with the **Input** statement to ensure that the variable list is consistent between the input and output routines.

10. Null characters are ignored.

Example

```
'This example creates a file called test.dat and writes a series
'of variables into it. Then the variables are read using the
'Input# function.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Open "test.dat" For Output As #1
   Write #1,2112,"David","McCue","123-45-6789"
   Close
   Open "test.dat" For Input As #1
   Input #1,x%,st1$,st2$,st3$
   message = "Employee " & x% & " Information" & crlf & crlf
   message = message & "First Name: " & st1$ & crlf
   message = message & "Last Name: "& st2$ & crlf
   message = message & "Social Security Number: " & sy3$
   MsgBox message
   Close
   Kill "test.dat"
End Sub
```

See Also **Open** (statement); **Get** (statement); **Line Input#** (statement); **Input**, **Input$, Input**B, **InputB$** (functions).

Platform(s) All.

Input, Input$, InputB, InputB$ (functions)

Syntax

```
Input[$](numchars,[#]filenumber)
InputB[$](numbytes,[#]filenumber)
```

Description Returns a specified number of characters or bytes read from a given sequential file.

Comments The **Input$** and **InputB$** functions return a **String**, whereas **Input** and **InputB** return a **String** variant.

The following parameters are required:

Parameter	Description
numchars	**Integer** containing the number of characters to be read from the file.
numbytes	**Integer** containing the number of bytes to be read from the file.

Parameter	Description
filenumber	**Integer** referencing a file opened in either **Input** or **Binary** mode. This is the same number passed to the **Open** statement.

The **Input** and **Input$** functions read all characters, including spaces and end-of-lines. Null characters are ignored.

The **InputB** and **InputB$** functions are used to read byte data from a file.

Example

```
'This example opens the autoexec.bat file and displays it in a
'dialog box.
Const crlf = Chr$(13) & Chr$(10)
Sub Main()
   x& = FileLen("c:\autoexec.bat")
   If x& > 0 Then
      Open "c:\autoexec.bat" For Input As #1
   Else
      MsgBox "File not found or empty."
      Exit Sub
   End If
   If x& > 80 Then
      ins = Input(80,#1)
   Else
      ins = Input(x,#1)
   End If
   Close
   MsgBox "File length: " & x& & crlf & ins
End Sub
```

See Also **Open** (statement); **Get** (statement); **Input#** (statement); **Line Input#** (statement).

Platform(s) All.

InputBox, InputBox$ (functions)

Syntax `InputBox[$](`*prompt* `[, [`*title*`] [, [`*default*`] [,[`*xpos*`],[`*ypos*`] [,`*helpfile*`,`*context*`]]]])`

Description Displays a dialog box with a text box into which the user can type.

Comments The content of the text box is returned as a **String** (in the case of **InputBox$**) or as a **String** variant (in the case of **InputBox**). A zero-length string is returned if the user selects Cancel.

The **InputBox**/**InputBox$** functions take the following named parameters:

Named Parameter	Description
prompt	Text to be displayed above the text box. The *prompt* parameter can contain multiple lines, each separated with an end-of-line (a carriage return, line feed, or carriage-return/line-feed pair). A runtime error is generated if *prompt* is **Null**.
title	Caption of the dialog box. If this parameter is omitted, then no title appears as the dialog box's caption. A runtime error is generated if *title* is **Null**.
default	Default response. This string is initially displayed in the text box. A runtime error is generated if *default* is **Null**.
xpos, *ypos*	**Integer** coordinates, given in twips (twentieths of a point), specifying the upper left corner of the dialog box relative to the upper left corner of the screen. If the position is omitted, then the dialog box is positioned on or near the application executing the script.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

You can type a maximum of 255 characters into **InputBox**.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 on most platforms). Invoking help does not remove the dialog.

When Cancel is selected, an empty string is returned. An empty string is also returned when the user selects the OK button with no text in the input box. Thus, it is not possible to determine the difference between these two situations. If you need to determine the difference, you should create a user-defined dialog or use the **AskBox** function.

Example

```
Sub Main()
   s$ = InputBox$("File to copy:","Copy","sample.txt")
End Sub
```

See Also **MsgBox** (statement); **AskBox, AskBox$** (functions); **AskPassword**, **AskPassword$** (function); **OpenFileName$** (function); **SaveFileName$** (function); **SelectBox** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, OS/2, Macintosh, UNIX.

InStr, InstrB (functions)

Syntax

```
InStr([start,] search, find [,compare])
InStrB([start,] search, find [,compare])
```

Description

Returns the first character position of string *find* within string *search*.

Comments

The **InStr** function takes the following parameters:

Parameter	Description
start	**Integer** specifying the character position (for **Instr**) or byte position (for **InstrB**) where searching begins. The *start* parameter must be between 1 and 32767. If this parameter is omitted, then the search starts at the beginning (*start* = 1).
search	Text to search. This can be any expression convertible to a **String**.
find	Text for which to search. This can be any expression convertible to a **String**.
compare	**Integer** controlling how string comparisons are performed. It can be any of the following values: 0 String comparisons are case-sensitive. 1 String comparisons are case-insensitive. Any other value produces a runtime error. If this parameter is omitted, then string comparisons use the current **Option Compare** setting. If no **Option Compare** statement has been encountered, then **Binary** is used (i.e., string comparisons are case-sensitive).

If the string is found, then its character position within *search* is returned, with 1 being the character position of the first character.

The **InStr** and **InStrB** functions observe the following additional rules:

- If either *search* or *find* is **Null**, then **Null** is returned.
- If the *compare* parameter is specified, then *start* must also be specified. In other words, if there are three parameters, then it is assumed that these parameters correspond to *start*, *search*, and *find*.
- A runtime error is generated if *start* is **Null**.
- A runtime error is generated if *compare* is not 0 or 1.
- If *search* is **Empty**, then 0 is returned.

- If *find* is **Empty**, then *start* is returned. If *start* is greater than the length of *search*, then 0 is returned.
- A runtime error is generated if *start* is less than or equal to zero.

The **InStr** and **InStrB** functions operate on character and byte data respectively. The Instr function interprets the *start* parameter as a character, performs a textual comparisons, and returns a character position. The **InStrB** function, on the other hand, interprets the *start* parameter as a byte position, performs binary comparisons, and returns a byte position.

On SBCS platforms, the **InStr** and **InStrB** functions are identical.

Example

```
'This example checks to see whether one string is in another
'and, if it is, then it copies the string to a variable and
'displays the result.
Sub Main()
   a$ = "This string contains the name Stuart."
   x% = InStr(a$,"Stuart",1)
   If x% <> 0 Then
      b$ = Mid$(a$,x%,6)
      MsgBox b$ & " was found."
      Exit Sub
   Else
      MsgBox "Stuart not found."
   End If
End Sub
```

See Also **Mid**, **Mid$**, **MidB**, **MidB$** (functions); **Option Compare** (statement); **Item$** (function); **Word$** (function); **Line$** (function).

Platform(s) All.

Int (function)

Syntax `Int(`*number*`)`

Description Returns the integer part of *number*.

Comments This function returns the integer part of a given value by returning the first integer less than the *number*. The sign is preserved.

The **Int** function returns the same type as *number*, with the following exceptions:

- If *number* is **Empty**, then an **Integer** variant of value 0 is returned.
- If *number* is a **String**, then a **Double** variant is returned.
- If *number* is **Null**, then a **Null** variant is returned.

Example

```
'This example extracts the integer part of a number.
```

```
Sub Main()
   a# = -1234.5224
   b% = Int(a#)
   MsgBox "The integer part of -1234.5224 is: " & b%
End Sub
```

See Also **Fix** (function); **CInt** (function).

Platform(s) All.

Integer (data type)

Syntax `Integer`

Description A data type used to declare whole numbers with up to four digits of precision.

Comments **Integer** variables are used to hold numbers within the following range:

`–32768 <= integer <= 32767`

Internally, integers are 2-byte short values. Thus, when appearing within a structure, integers require 2 bytes of storage. When used with binary or random files, 2 bytes of storage are required.

When passed to external routines, **Integer** values are sign-extended to the size of an integer on that platform (either 16 or 32 bits) before pushing onto the stack.

The type-declaration character for **Integer** is %.

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CInt** (function).

Platform(s) All.

IPmt (function)

Syntax `IPmt(`*rate*`, `*per*`, `*nper*`, `*pv*`, `*fv*`, `*due*`)`

Description Returns the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Comments An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages, monthly savings plans, and retirement plans.

The following table describes the named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period. If the payment periods are monthly, be sure to divide the annual interest rate by 12 to get the monthly rate.
per	**Double** representing the payment period for which you are calculating the interest payment. If you want to know the interest paid or received during period 20 of an annuity, this value would be 20.
nper	**Double** representing the total number of payments in the annuity. This is usually expressed in months, and you should be sure that the interest rate given above is for the same period that you enter here.
pv	**Double** representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan because that is the amount of cash you have in the present. In the case of a retirement plan, this value would be the current value of the fund because you have a set amount of principal in the plan.
fv	**Double** representing the future value of your annuity. In the case of a loan, the future value would be zero because you will have paid it off. In the case of a savings plan, the future value would be the balance of the account after all payments are made.
due	**Integer** indicating when payments are due. If this parameter is 0, then payments are due at the end of each period (usually, the end of the month). If this value is 1, then payments are due at the start of each period (the beginning of the month).

The *rate* and *nper* parameters must be in expressed in the same units. If *rate* is expressed in percentage paid per month, then *nper* must also be expressed in months. If *rate* is an annual rate, then the period given in *nper* should also be in years or the annual *rate* should be divided by 12 to obtain a monthly rate.

If the function returns a negative value, it represents interest you are paying out, whereas a positive value represents interest paid to you.

Example

```
'This example calculates the amount of interest paid on a
'$1,000.00 loan financed over 36 months with an annual interest
'rate of 10%. Payments are due at the beginning of the month.
'The interest paid during the first 10 months is displayed in a
'table.
```

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   For x = 1 to 10
      ipm# = IPmt((.10/12),x,36,1000,0,1)
      message = message & Format(x,"00") & " : " _
         & Format(ipm#," 0,0.00") & crlf
   Next x
   MsgBox message
End Sub
```

See Also **NPer** (function); **Pmt** (function); **PPmt** (function); **Rate** (function).

Platform(s) All.

IRR (function)

Syntax `IRR(`*valuearray()*`,`*guess*`)`

Description Returns the internal rate of return for a series of periodic payments and receipts.

Comments The internal rate of return is the equivalent rate of interest for an investment consisting of a series of positive and/or negative cash flows over a period of regular intervals. It is usually used to project the rate of return on a business investment that requires a capital investment up front and a series of investments and returns on investment over time.

The IRR function requires the following named parameters:

Named Parameter	Description
valuearray()	Array of **Double** numbers that represent payments and receipts. Positive values are payments, and negative values are receipts. There must be at least one positive and one negative value to indicate the initial investment (negative value) and the amount earned by the investment (positive value).
guess	**Double** containing your guess as to the value that the **IRR** function will return. The most common guess is .1 (10 percent).

The value of **IRR** is found by iteration. It starts with the value of *guess* and cycles through the calculation adjusting *guess* until the result is accurate within 0.00001 percent. After 20 tries, if a result cannot be found, **IRR** fails, and the user must pick a better guess.

Example

```
'This example illustrates the purchase of a lemonade stand for
'$800 and a series of incomes from the sale of lemonade over 12
'months. The projected incomes for this example are generated
'in two For...Next Loops, and then the internal rate of return
```

```
'is calculated and displayed. (Not a bad investment!)
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim valu#(12)
   valu(1) = -800    'Initial investment
   message = valu#(1) & ", "
   'Calculate the second through fifth months' sales.
   For x = 2 To 5
      valu(x) = 100 + (x * 2)
      message = message & valu(x) & ", "
   Next x
   'Calcluate the sixth through twelfth months' sales.
   For x = 6 To 12
      valu(x) = 100 + (x * 10)
      message = message & valu(x) & ", "
   Next x
   'Calcluate the equivalent investment return rate.
   retrn# = IRR(valu,.1)
   message = "The values: " & crlf & message & crlf & crlf
   MsgBox message & "Return rate: " & Format(retrn#,"Percent")
End Sub
```

See Also **Fv** (function); **MIRR** (function); **Npv** (function); **Pv** (function).

Platform(s) All.

Is (operator)

Syntax *object* `Is [`*object* `| Nothing]`

Description Returns **True** if the two operands refer to the same object; returns **False** otherwise.

Comments This operator is used to determine whether two object variables refer to the same object. Both operands must be object variables of the same type (i.e., the same data object type or both of type **Object**).

The **Nothing** constant can be used to determine whether an object variable is uninitialized:

```
If MyObject Is Nothing Then MsgBox "MyObject is
uninitialized."
```

Uninitialized object variables reference no object.

Example

```
'This function inserts the date into a Microsoft Word document.
Sub InsertDate(ByVal WinWord As Object)
   If WinWord Is Nothing Then
      MsgBox "Object variant is not set."
   Else
```

```
      WinWord.Insert Date$
   End If
End Sub
Sub Main()
   Dim WinWord As Object
   On Error Resume Next
   WinWord = CreateObject("word.basic")
   InsertDate WinWord
End Sub
```

See Also Operator Precedence (topic); **Like** (operator).

Platform(s) All.

Platform Notes **Windows, Win32, Macintosh:** When comparing OLE Automation objects, the **Is** operator will only return **True** if the operands reference the same OLE Automation object. This is different from data objects. For example, the following use of **Is** (using the object class called **excel.application**) returns **True**:

```
Dim a As Object
Dim b As Object
a = CreateObject("excel.application")
b = a
If a Is b Then Beep
```

The following use of **Is** will return **False**, even though the actual objects may be the same:

```
Dim a As Object
Dim b As Object
a = CreateObject("excel.application")
b = GetObject(,"excel.application")
If a Is b Then Beep
```

The **Is** operator may return **False** in the above case because, even though a and b reference the same object, they may be treated as different objects by OLE 2.0 (this is dependent on the OLE 2.0 server application).

IsDate (function)

Syntax `IsDate(`*expression*`)`

Description Returns **True** if *expression* can be legally converted to a date; returns **False** otherwise.

Example

```
Sub Main()
   Dim a As Variant
Retry:
   a = InputBox("Enter a date.", "Enter Date")
   If IsDate(a) Then
```

```
      MsgBox Format(a,"long date")
   Else
      Msgbox "Not quite, please try again!"
      Goto Retry
   End If
End Sub
```

See Also **Variant** (data type); **IsEmpty** (function); **IsError** (function); **IsObject** (function); **VarType** (function); **IsNull** (function).

Platform(s) All.

IsEmpty (function)

Syntax `IsEmpty(`*expression*`)`

Description Returns **True** if *expression* is a **Variant** variable that has never been initialized; returns **False** otherwise.

Comments The **IsEmpty** function is the same as the following:

```
(VarType(expression) = ebEmpty)
```

Example

```
Sub Main()
   Dim a As Variant
   If IsEmpty(a) Then
      a = 1.0#    'Give uninitialized data a Double value 0.0.
      MsgBox "The variable has been initialized to: " & a
   Else
      MsgBox "The variable was already initialized!"
   End If
End Sub
```

See Also **Variant** (data type); **IsDate** (function); **IsError** (function); **IsObject** (function); **VarType** (function); **IsNull** (function).

Platform(s) All.

IsError (function)

Syntax `IsError(`*expression*`)`

Description Returns **True** if expression is a user-defined error value; returns **False** otherwise.

Example

```
'This example creates a function that divides two numbers. If
'there is an error dividing the numbers, then a variant of type
'"error" is returned. Otherwise, the function returns the result
'of the division. The IsError function is used to determine
```

```
'whether the function encountered an error.
Function Div(ByVal a,ByVal b) As Variant
   If b = 0 Then
      Div = CVErr(2112)'Return a special error value.
   Else
      Div = a / b 'Return the division.
   End If
End Function
Sub Main()
   Dim a As Variant
   a = Div(10,12)
   If IsError(a) Then
      MsgBox "The following error occurred: " & CStr(a)
   Else
      MsgBox "The result is: " & a
   End If
End Sub
```

See Also **Variant** (data type); **IsEmpty** (function); **IsDate** (function); **IsObject** (function); **VarType** (function); **IsNull** (function).

Platform(s) All.

IsMissing (function)

Syntax `IsMissing(`*argname*`)`

Description Returns **True** if *argname* was passed to the current subroutine or function; returns **False** if omitted.

Comments The **IsMissing** function is used with variant variables passed as optional parameters (using the **Optional** keyword) to the current subroutine or function. For nonvariant variables or variables that were not declared with the **Optional** keyword, **IsMissing** will always return **True**.

Example

```
'The following function runs an application and optionally
'minimizes it. If the optional isMinimize parameter is not
'specified by the caller, then the application is not minimized.
Sub Test(AppName As String,Optional isMinimize As Variant)
   app = Shell(AppName)
   If Not IsMissing(isMinimize) Then
      AppMinimize app
   Else
      AppMaximize app
   End If
End Sub
Sub Main
   Test "Notepad"'Maximize this application
```

```
   Test "Notepad",True'Mimimize this application
End Sub
```

See Also **Declare** (statement); **Sub...End Sub** (statement); **Function...End Function** (statement).

Platform(s) All.

IsNull (function)

Syntax `IsNull(`*expression*`)`

Description Returns **True** if *expression* is a **Variant** variable that contains no valid data; returns **False** otherwise.

Comments The **IsNull** function is the same as the following:

```
(VarType(expression) = ebNull)
```

Example

```
Sub Main()
   Dim a As Variant'Initialized as Empty
   If IsNull(a) Then MsgBox "The variable contains no valid
data."
   a = Empty * Null
   If IsNull(a) Then MsgBox "Null propagated through the
expression."
End Sub
```

See Also **Variant** (data type); **IsEmpty** (function); **IsDate** (function); **IsError** (function); **IsObject** (function); **VarType** (function).

Platform(s) All.

IsNumeric (function)

Syntax `IsNumeric(`*expression*`)`

Description Returns **True** if *expression* can be converted to a number; returns **False** otherwise.

Comments If passed a number or a variant containing a number, then **IsNumeric** always returns **True**.

If a **String** or **String** variant is passed, then **IsNumeric** will return **True** only if the string can be converted to a number. The following syntaxes are recognized as valid numbers:

`&H`*hexdigits*`[&|%|!|#|@]`

`&[O]`*octaldigits*`[&|%|!|#|@]`

`[-|+]`*digits*`[.[`*digits*`]][E[-|+]`*digits*`][!|%|&|#|@]`

If an **Object** variant is passed, then the default property of that object is retrieved and one of the above rules is applied.

IsNumeric returns **False** if *expression* is a **Date**.

Example

```
Sub Main()
   Dim s$ As String
   s$ = InputBox("Enter a number.","Enter Number")
   If IsNumeric(s$) Then
      MsgBox "You did good!"
   Else
      MsgBox "You didn't do so good!"
   End If
End Sub
```

See Also **Variant** (data type); **IsEmpty** (function); **IsDate** (function); **IsError** (function); **IsObject** (function); **VarType** (function); **IsNull** (function).

Platform(s) All.

IsObject (function)

Syntax `IsObject(`*expression*`)`

Description Returns **True** if *expression* is a **Variant** variable containing an **Object**; returns **False** otherwise.

Example

```
'This example will attempt to find a running copy of Excel and
'create an Excel object that can be referenced as any other
'object in BasicScript.
Sub Main()
   Dim v As Variant
   On Error Resume Next
   Set v = GetObject(,"Excel.Application")
   If IsObject(v) Then
      MsgBox"The default object value is: " & v = v.Value'Access
value property of the object.
   Else
      MsgBox "Excel not loaded."
   End If
End Sub
```

See Also **Variant** (data type); **IsEmpty** (function); **IsDate** (function); **IsError** (function); **VarType** (function); **IsNull** (function).

Platform(s) All.

Item$ (function)

Syntax `Item$(`*text$*`,` *first* `[,[`*last*`] [,`*delimiters$*`]])`

Description Returns all the items between *first* and *last* within the specified formatted text list.

Comments The **Item$** function takes the following parameters:

Parameter	Description
text$	**String** containing the text from which a range of items is returned.
first	**Integer** containing the index of the first item to be returned. If *first* is greater than the number of items in *text$*, then a zero-length string is returned.
last	**Integer** containing the index of the last item to be returned. All of the items between *first* and *last* are returned. If *last* is greater than the number of items in *text$*, then all items from *first* to the end of text are returned. If *last* is missing, then only the item specified by *first* is returned. An "Invalid use of Null" error is returned if this parameter is **Null**.
delimiters$	**String** containing different item delimiters. By default, items are separated by commas and end-of-lines. This can be changed by specifying different delimiters in the *delimiters$* parameter.

The **Item$** function treats embedded null characters as regular characters.

An empty string is returned if *first* is less than 1. If *last* is less than *first*, the values are swapped.

Example

```
'This example creates two delimited lists and extracts a range
'from each, then displays the result in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   ilist$ = "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15"
   slist$ = "1/2/3/4/5/6/7/8/9/10/11/12/13/14/15"
   list1$ = Item$(ilist$,5,12)
   list2$ = Item$(slist$,2,9,"/")
   MsgBox "The returned lists are: " & crlf & list1$ & crlf &
list2$
End Sub
```

See Also **ItemCount** (function); **Line$** (function); **LineCount** (function); **Word$** (function); **WordCount** (function).

Platform(s) All.

ItemCount (function)

Syntax `ItemCount(`*text$* `[,`*delimiters$*`])`

Description Returns an **Integer** containing the number of items in the specified delimited text.

Comments Items are substrings of a delimited text string. Items, by default, are separated by commas and/or end-of-lines. This can be changed by specifying different delimiters in the *delimiters$* parameter. For example, to parse items using a backslash:

```
n = ItemCount(text$,"\")
```

The **ItemCount** function treats embedded null characters as regular characters.

Example

```
'This example creates two delimited lists and then counts the
'number of items in each. The counts are displayed in a dialog
'box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   ilist$ = "1,2,3,4,5,6,7,8,9,10,11,12,13,14,15"
   slist$ = "1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19"
   l1% = ItemCount(ilist$)
   l2% = ItemCount(slist$,"/")
   message = "The first lists contains: " & l1% & " items." & crlf
   message = message & "The second list contains: " _
      & l2% & " items."
   MsgBox message
End Sub
```

See Also **Item$** (function); **Line$** (function); **LineCount** (function); **Word$** (function); **WordCount** (function).

Platform(s) All.

Keywords (topic)

A keyword is any word or symbol recognized by BasicScript as part of the language. All of the following are keywords:

Access	**Alias**	**And**	**Any**
Append	**As**	**Base**	**Begin**
Binary	**Boolean**	**ByRef**	**ByVal**
Call	**CancelButton**	**Case**	**CDecl**
CheckBox	**Chr**	**ChrB**	**ChrW**
Close	**ComboBox**	**Compare**	**Const**
CStrings	**Currency**	**Date**	**Declare**
Default	**DefBool**	**DefCur**	**DefDate**
DefDbl	**DefInt**	**DefLng**	**DefObj**
DefSng	**DefStr**	**DefVar**	**Dialog**
Dim	**Do**	**Double**	**DropListBox**
Else	**ElseIf**	**End**	**Eqv**
Error	**Exit**	**Explicit**	**For**
Function	**Get**	**Global**	**GoSub**
Goto	**GroupBox**	**HelpButton**	**If**
Imp	**Inline**	**Input**	**Input**
InputB	**Integer**	**Is**	**Len**
Let	**Lib**	**Like**	**Line**
ListBox	**Lock**	**Long**	**Loop**
LSet	**Mid**	**MidB**	**Mod**
Name	**New**	**Next**	**Not**
Nothing	**Object**	**Off**	**OKButton**
On	**Open**	**Option**	**Optional**

OptionButton	**OptionGroup**	**Or**	**Output**
ParamArray	**Pascal**	**Picture**	**PictureButton**
Preserve	**Print**	**Private**	**Public**
PushButton	**Put**	**Random**	**Read**
ReDim	**Rem**	**Resume**	**Return**
RSet	**Seek**	**Select**	**Set**
Shared	**Single**	**Spc**	**Static**
StdCall	**Step**	**Stop**	**String**
Sub	**System**	**Tab**	**Text**
TextBox	**Then**	**Time**	**To**
Type	**Unlock**	**Until**	**Variant**
WEnd	**While**	**Width**	**Write**
Xor			

Restrictions

All keywords are reserved by BasicScript, in that you cannot create a variable, function, constant, or subroutine with the same name as a keyword. However, you are free to use all keywords as the names of structure members.

For all other keywords in BasicScript (such as **MsgBox**, **Str**, and so on), the following restrictions apply:

- You can create a subroutine or function with the same name as a keyword.
- You can create a variable with the same name as a keyword as long as the variable is first explicitly declared with a **Dim**, **Private**, or **Public** statement.

Platform(s) All.

Kill (statement)

Syntax

```
Kill pathname
Kill pathname [,filetype]
Kill filetype
```

Description Deletes all files matching *pathname*.

Comments The **Kill** statement accepts the following named parameters:

Named Parameter	Description
pathname	Specifies the file to delete. If *filetype* is specified, then this parameter must specify a path. Otherwise, this parameter can include both a path and a file specification containing wildcards.
filetype	Specifies the type of file on a Macintosh. If *pathname* is also specified, it indicates the directory from which files will be removed. Otherwise, files are removed from the current directory. File types are specified using the MacID function.

The *pathname* argument can include wildcards, such as * and ?. The * character matches any sequence of zero or more characters, whereas the ? character matches any single character. Multiple *'s and ?'s can appear within the expression to form complex searching patterns. The following table shows some examples.

This Pattern	Matches These Files	Doesn't Match These Files
*S.*TXT	SAMPLE. TXT	SAMPLE
	GOOSE.TXT	SAMPLE.DAT
	SAMS.TXT	
C*T.TXT	CAT.TXT	CAP.TXT
		ACATS.TXT
C*T	CAT	CAT.DOC
	CAP.TXT	
C?T	CAT	CAT.TXT
	CUT	CAPIT
		CT
*	(All files)	

Example

```
'This example looks to see whether file test1.dat exists. If it
'does not, then it creates both test1.dat and test2.dat. The
'existence of the files is tested again; if they exist, a
'message is generated, and then they are deleted. The final test
'looks to see whether they are still there and displays the
'result.
Sub Main()
   If Not FileExists("test1.dat") Then
      Open "test1.dat" For Output As #1
      Open "test2.dat" For Output As #2
```

```
      Close
   End If
   If FileExists ("test1.dat") Then
      MsgBox "File test1.dat exists."
      Kill "test?.dat"
   End If
   If FileExists ("test1.dat") Then
      MsgBox "File test1.dat still exists."
   Else
      MsgBox "test?.dat sucessfully deleted."
   End If
End Sub
```

See Also **Name** (statement).

Platform(s) All.

Platform Notes **Windows:** For compatibility with DOS wildcard matching, BasicScript special-cases the pattern "*.*" to indicate all files, not just files with a periods in their names.

This function behaves the same as the "del" command in DOS.

Macintosh: The Macintosh does not support wildcard characters such as * and ?. These are valid filename characters. Instead of wildcards, the Macintosh uses the **MacID** function to specify a collection of files of the same type. The syntax for this function is:

```
Kill MacID(text$)
```

The *text$* parameter is a four-character string containing a file type, a resource type, an application signature, or an Apple event. A runtime error occurs if the **MacID** function is used on platforms other than the Macintosh.

LBound (function)

Syntax `LBound(ArrayVariable() [,dimension])`

Description Returns an **Integer** containing the lower bound of the specified dimension of the specified array variable.

Comments The *dimension* parameter is an integer specifying the desired dimension. If this parameter is not specified, then the lower bound of the first dimension is returned.

The **LBound** function can be used to find the lower bound of a dimension of an array returned by an OLE Automation method or property:

```
LBound(object.property [,dimension])
LBound(object.method [,dimension])
```

Examples

```
Sub Main()
   'This example dimensions two arrays and displays their lower
   'bounds.
```

```
   Dim a(5 To 12)
   Dim b(2 To 100, 9 To 20)
   lba = LBound(a)
   lbb = LBound(b,2)
   MsgBox "The lower bound of a is: " & lba _
      & " The lower bound of b is: " & lbb
   'This example uses LBound and UBound to dimension a dynamic
   'array to hold a copy of an array redimmed by the FileList
   'statement.
   Dim fl$()
   FileList fl$,"*.*"
   count = UBound(fl$)
   If ArrayDims(a) Then
      Redim nl$(LBound(fl$) To UBound(fl$))
      For x = 1 To count
         nl$(x) = fl$(x)
      Next x
      MsgBox "The last element of the new array is: " & nl$(count)
   End If
End Sub
```

See Also **UBound** (function); **ArrayDims** (function); Arrays (topic).

Platform(s) All.

LCase, LCase$ (functions)

Syntax `LCase[$](`*string*`)`

Description Returns the lowercase equivalent of the specified string.

Comments **LCase$** returns a **String**, whereas **LCase** returns a **String** variant.

Null is returned if *string* is **Null**.

Example

```
'This example shows the LCase function used to change uppercase
'names to lowercase with an uppercase first letter.
Sub Main()
   lname$ = "WILLIAMS"
   fl$ = Left$(lname$,1)
   rest$ = Mid$(lname$,2,Len(lname$))
   lname$ = fl$ & LCase$(rest$)
   MsgBox "The converted name is: " & lname$
End Sub
```

See Also **UCase**, **UCase$** (functions).

Platform(s) All.

Left, Left$, LeftB, LeftB$ (functions)

Syntax

```
Left[$](string, length)
LeftB[$](string,length)
```

Description

Returns the leftmost *length* characters (for **Left** and **Left$**) or bytes (for **LeftB** and **LeftB$**) from a given string.

Comments

Left$ returns a **String**, whereas **Left** returns a **String** variant.

The *length* parameter is an **Integer** value specifying the number of characters to return. If *length* is 0, then a zero-length string is returned. If *length* is greater than or equal to the number of characters in the specified string, then the entire string is returned.

The **LeftB** and **LeftB$** functions are used to return a sequence of bytes from a string containing byte data. In this case, *length* specifies the number of bytes to return. If *length* is greater than the number of bytes in *string*, then the entire string is returned.

Null is returned if *string* is **Null**.

Example

```
'This example shows the Left$ function used to change uppercase
'names to lowercase with an uppercase first letter.
Sub Main()
   lname$ = "WILLIAMS"
   fl$ = Left$(lname$,1)
   rest$ = Mid$(lname$,2,Len(lname$))
   lname$ = fl$ & LCase$(rest$)
   MsgBox "The converted name is: " & lname$
End Sub
```

See Also

Right, **Right$, RightB**, **RightB$** (functions).

Platform(s)

All.

Len, LenB (functions)

Syntax

```
Len(expression)
LenB(expression)
```

Description

Returns the number of characters (for **Len**) or bytes (for **LenB**) in **String** expression or the number of bytes required to store the specified variable.

Comments

If *expression* evaluates to a String, then **Len** returns the number of characters in a given string or 0 if the string is empty. When used with a **Variant** variable, the length of the variant when converted to a **String** is returned. If *expression* is a **Null**, then **Len** returns a **Null** variant.

The **LenB** function is used to return the number of bytes in a given string. On SBCS systems, the **LenB** and **Len** functions are identical.

If used with a non-**String** or non-**Variant** variable, these functions returns the number of bytes occupied by that data element.

When used with user-defined data types, these functions return the combined size of each member within the structure. Since variable-length strings are stored elsewhere, the size of each variable-length string within a structure is 2 bytes.

The following table describes the sizes of the individual data elements when appearing within a structure:

Data Element	Size
Integer	2 bytes.
Long	4 bytes.
Float	4 bytes.
Double	8 bytes.
Currency	8 bytes.
String (variable-length)	2 bytes
String (fixed-length)	The length of the string as it appears in the string's declaration in characters for **Len** and bytes for **LenB**.
Objects	0 bytes. Both data object variables and variables of type **Object** are always returned as 0 size.
User-defined type	Combined size of each structure member. Variable-length strings within structures require 2 bytes of storage. Arrays within structures are fixed in their dimensions. The elements for fixed arrays are stored within the structure and therefore require the number of bytes for each array element multiplied by the size of each array dimension: `element_size*`*`dimension1*dimension2...`*

The **Len** and **LenB** functions always returns 0 with object variables or any data object variable.

Examples

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   'This example shows the Len function used in a routine to
   'change uppercase names to lowercase with an uppercase first
   'letter.
   lname$ = "WILLIAMS"
   fl$ = Left$(lname$,1)
   ln% = Len(lname$)
   rest$ = Mid$(lname$,2,ln%)
```

```
   lname$ = fl$ & LCase$(rest$)
   MsgBox "The converted name is: " & lname$
   'This example returns a table of lengths for standard numeric
   'types.
   Dim lns(4)
   a% = 100 : b& = 200 : c! = 200.22 : d# = 300.22
   lns(1) = Len(a%)
   lns(2) = Len(b&)
   lns(3) = Len(c!)
   lns(4) = Len(d#)
   message = "Lengths of standard types:" & crlf
   message = message & "Integer: " & lns(1) & crlf
   message = message & "Long: " & lns(2) & crlf
   message = message & "Single: " & lns(3) & crlf
   message = message & "Double: " & lns(4) & crlf
   MsgBox message
End Sub
```

See Also **InStr, InStrB** (functions).

Platform(s) All.

Let (statement)

Syntax `[Let]` *variable* `=` *expression*

Description Assigns the result of an expression to a variable.

Comments The use of the word **Let** is supported for compatibility with other implementations of BasicScript. Normally, this word is dropped.

When assigning expressions to variables, internal type conversions are performed automatically between any two numeric quantities. Thus, you can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This happens when the larger type contains a numeric quantity that cannot be represented by the smaller type. For example, the following code will produce a runtime error:

```
Dim amount As Long
Dim quantity As Integer
amount = 400123   'Assign a value out of range for int.
quantity = amount 'Attempt to assign to Integer.
```

When performing an automatic data conversion, underflow is not an error.

Example

```
Sub Main()
   Let a$ = "This is a string."
   Let b% = 100
   Let c# = 1213.3443
```

```
End Sub
```

See Also = (operator); Expression Evaluation (topic).

Platform(s) All.

Like (operator)

Syntax *expression* `Like` *pattern*

Description Compares two strings and returns **True** if the *expression* matches the given pattern; returns **False** otherwise.

Comments Case sensitivity is controlled by the **Option Compare** setting.

The pattern expression can contain special characters that allow more flexible matching:

Character	Evaluates To
?	Matches a single character.
*	Matches one or more characters.
#	Matches any digit.
[*range*]	Matches if the character in question is within the specified range.
[!*range*]	Matches if the character in question is not within the specified range.

A *range* specifies a grouping of characters. To specify a match of any of a group of characters, use the syntax **[ABCDE]**. To specify a range of characters, use the syntax **[A-Z]**. Special characters must appear within brackets, such as **[]*?#**.

If *expression* or *pattern* is not a string, then both *expression* and *pattern* are converted to **String** variants and compared, returning a **Boolean** variant. If either variant is **Null**, then **Null** is returned.

The following table shows some examples:

expression	**True If** *pattern* **Is**	**False If** *pattern* **Is**
"EBW"	"E*W", "E*"	"E*B"
"BasicScript"	"B*[r-t]icScript"	"B[r-t]ic"
"Version"	"V[e]?s*n"	"V[r]?s*N"
"2.0"	"#.#","#?#"	"###","#?[!0-9]"
"[ABC]"	"[[]*]"	"[ABC]","[*]"

Example

```
'This example demonstrates various uses of the Like function.
Sub Main()
```

```
   a$ = "This is a string variable of 123456 characters"
   b$ = "123.45"
   If a$ Like "[A-Z][g-i]*" Then _
      MsgBox "The first comparison is True."
   If b$ Like "##3.##" Then _
      MsgBox "The second comparison is True."
   If a$ Like "*variable*" Then _
      MsgBox "The third comparison is True."
End Sub
```

See Also Operator Precedence (topic); **Is** (operator); **Option Compare** (statement).

Platform(s) All.

Line Input# (statement)

Syntax `Line Input [#]`*filenumber*`,`*variable*

Description Reads an entire line into the given variable.

Comments The *filenumber* parameter is a number that is used by BasicScript to refer to the open file—the number passed to the **Open** statement. The *filenumber* must reference a file opened in **Input** mode.

The file is read up to the next end-of-line, but the end-of-line character(s) is (are) not returned in the string. The file pointer is positioned after the terminating end-of-line.

The *variable* parameter is any string or variant variable reference. This statement will automatically declare the variable if the specified variable has not yet been used or dimensioned.

This statement recognizes either a single line feed or a carriage-return/line-feed pair as the end-of-line delimiter.

A runtime error is generated if you attempt to read beyond the end of the file.

Example

```
'This example reads five lines of the autoexec.bat file and
'displays them in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Open "c:\autoexec.bat" For Input As #1
   For x = 1 To 5
      Line Input #1,lin$
      message = message & lin$ & crlf
   Next x
   MsgBox "The first 5 lines of your autoexec.bat are:" & crlf &
Msg
End Sub
```

See Also **Open** (statement); **Get** (statement); **Input#** (statement); **Input**, **Input$, InputB**, **InputB$** (functions).

Platform(s) All.

Line Numbers (topic)

Line numbers are not supported by BasicScript.

As an alternative to line numbers, you can use meaningful labels as targets for absolute jumps, as shown below:

```
Sub Main()
   Dim i As Integer
   On Error Goto MyErrorTrap
   i = 0
LoopTop:
   i = i + 1
   If i < 10 Then Goto LoopTop
MyErrorTrap:
   MsgBox "An error occurred."
End Sub
```

Line$ (function)

Syntax `Line$(`*text$*`,` *first*`[,` *last*`])`

Description Returns a **String** containing a single line or a group of lines between *first* and *last*.

Comments Lines are delimited by carriage return, line feed, or carriage-return/line-feed pairs. Embedded null characters are treated as regular characters.

The **Line$** function takes the following parameters:

Parameter	Description
text$	**String** containing the text from which the lines will be extracted.
first	**Integer** representing the index of the first line to return. If *last* is omitted, then this line will be returned. If *first* is greater than the number of lines in *text$*, then a zero-length string is returned.
last	**Integer** representing the index of the last line to return

Example

```
'This example reads five lines of the autoexec.bat file,
```

```
'extracts the third and fourth lines with the Line$ function,
'and displays them in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Open "c:\autoexec.bat" For Input As #1
   For x = 1 To 5
      Line Input #1,lin$
      txt = txt & lin$ & crlf
   Next x
   lines$ = Line$(txt,3,4)
   MsgBox lines$
End Sub
```

See Also **Item$** (function); **ItemCount** (function); **LineCount** (function); **Word$** (function); **WordCount** (function).

Platform(s) All.

LineCount (function)

Syntax `LineCount(`*text$*`)`

Description Returns an **Integer** representing the number of lines in *text$*.

Comments Lines are delimited by carriage return, line feed, or both. Embedded null characters are treated as regular characters.

Example

```
'This example reads the first ten lines of your autoexec.bat
'file, uses the LineCount function to determine the number of
'lines, and then displays them in a message box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   x = 1
   Open "c:\autoexec.bat" For Input As #1
   While (x < 10) And Not EOF(1)
      Line Input #1,lin$
      txt = txt & lin$ & crlf
      x = x + 1
   Wend
   lines! = LineCount(txt)
   MsgBox "The number of lines in txt is: " _
      & lines! & crlf & crlf & txt
End Sub
```

See Also **Item$** (function); **ItemCount** (function); **Line$** (function); **Word$** (function); **WordCount** (function).

Platform(s) All.

ListBox (statement)

Syntax `ListBox` *x*,*y*,*width*,*height*,*ArrayVariable*,*.Identifier*

Description Creates a list box within a dialog box template.

Comments When the dialog box is invoked, the list box will be filled with the elements contained in *ArrayVariable*.

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **ListBox** statement requires the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** coordinates specifying the dimensions of the control in dialog units.
ArrayVariable	Specifies a single-dimensioned array of strings used to initialize the elements of the list box. If this array has no dimensions, then the list box will be initialized with no elements. A runtime error results if the specified array contains more than one dimension. *ArrayVariable* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the index of the list box's selection (0 is the first item, 1 is the second, and so on). This variable can be accessed using the following syntax: *DialogVariable*.*Identifier*

Example

```
'This example creates a dialog box with two list boxes, one
'containing files and the other containing directories.
Sub Main()
   Dim files() As String
   Dim dirs() As String
   Begin Dialog ListBoxTemplate 16,32,184,96,"Sample"
      Text 8,4,24,8,"&Files:"
      ListBox 8,16,60,72,files$,.Files
      Text 76,4,21,8,"&Dirs:"
```

```
      ListBox 76,16,56,72,dirs$,.Dirs
      OKButton 140,4,40,14
      CancelButton 140,24,40,14
   End Dialog
   FileList files
   FileDirs dirs
   Dim ListBoxDialog As ListBoxTemplate
   rc% = Dialog(ListBoxDialog)
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

ListBoxEnabled (function)

Syntax `ListBoxEnabled(`*name$* | *id*`)`

Description Returns **True** if the given list box is enabled within the active window or dialog box; returns **False** otherwise.

Comments This function is used to determine whether a list box is enabled within the current window or dialog box. If there is no active window, **False** will be returned.

The **ListBoxEnabled** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the list box. The name of a list box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a list box. A runtime error is generated if a list box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the list box.

Note: The **ListBoxEnabled** function is used to determine whether a list box is enabled in another application's dialog box. Use the **DlgEnable** function in dynamic dialog boxes.

Example

```
'This example checks to see whether the list box is enabled
'before setting the focus to it.
Sub Main()
```

```
   If ListBoxEnabled("Files:") Then ActivateControl "Files:"
End Sub
```

See Also **GetListBoxItem$** (function); **GetListBoxItemCount** (function); **ListBoxExists** (function); **SelectListBoxItem** (statement).

Platform(s) Windows.

ListBoxExists (function)

Syntax `ListBoxExists(`*name$* `|` *id*`)`

Description Returns **True** if the given list box exists within the active window or dialog box; returns **False** otherwise.

Comments This function is used to determine whether a list box exists within the current window or dialog box. If there is no active window, **False** will be returned.

The **ListBoxExists** function takes the following parameters:

Parameter	Description
name$	**String** containing the name of the list box. The name of a list box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a list box. A runtime error is generated if a list box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the list box.

Note: The **ListBoxExists** function is used to determine whether a list box exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This example checks to see whether the list box exists and is
'enabled before setting the focus to it.
Sub Main()
   If ListBoxExists("Files:") Then
      If ListBoxEnabled("Files:") Then
         ActivateControl "Files:"
      End If
   End If
End Sub
```

See Also **GetListBoxItem$** (function); **GetListBoxItemCount** (function); **ListBoxEnabled** (function); **SelectListBoxItem** (statement).

Platform(s) Windows.

Literals (topic)

Literals are values of a specific type. The following table shows the different types of literals supported by BasicScript:

Literal	Description
10	**Integer** whose value is 10.
43265	**Long** whose value is 43,265.
5#	**Double** whose value is 5.0. A number's type can be explicitly set using any of the following type-declaration characters: % **Integer** & **Long** # **Double** ! **Single**
5.5	**Double** whose value is 5.5. Any number with decimal point is considered a double.
5.4E100	**Double** expressed in scientific notation.
&HFF	**Integer** expressed in hexadecimal.
&O47	**Integer** expressed in octal.
&HFF#	**Double** expressed in hexadecimal.
"hello"	String of five characters: hello.
"""hello"""	String of seven characters: "hello". Quotation marks can be embedded within strings by using two consecutive quotation marks.
#1/1/1994#	Date value whose internal representation is 34335.0. Any valid date can appear with #'s. Date literals are interpreted at execution time using the locale settings of the host environment. To ensure that date literals are correctly interpreted for all locales, use the international date format: *YYYY-MM-DD HH:MM:SS#*

Constant Folding

BasicScript supports constant folding where constant expressions are calculated by the compiler at compile time. For example, the expression

```
i% = 10 + 12
```

is the same as:

```
i% = 22
```

Similarly, with strings, the expression

```
s$ = "Hello," + " there" + Chr(46)
```

is the same as:

```
s$ = "Hello, there."
```

Loc (function)

Syntax `Loc(`*filenumber*`)`

Description Returns a **Long** representing the position of the file pointer in the given file.

Comments The *filenumber* parameter is an **Integer** used by BasicScript to refer to the number passed by the **Open** statement to BasicScript.

The **Loc** function returns different values depending on the mode in which the file was opened:

File Mode	Returns
Input	Current byte position divided by 128
Output	Current byte position divided by 128
Append	Current byte position divided by 128
Binary	Position of the last byte read or written
Random	Number of the last record read or written

Example

```
'This example reads five lines of the autoexec.bat file,
'determines the current location of the file pointer, and
'displays it in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Open "c:\autoexec.bat" For Input As #1
   For x = 1 To 5
      If Not EOF(1) Then Line Input #1,lin$
   Next x
   lc% = Loc(1)
   Close
   MsgBox "The file location is: " & lc%
End Sub
```

See Also **Seek** (function); **Seek** (statement); **FileLen** (function).

Platform(s) All.

Lock, Unlock (statements)

Syntax

```
Lock [#] filenumber [,{record | [start] To end}]
Unlock [#] filenumber [,{record | [start] To end}]
```

Description

Locks or unlocks a section of the specified file, granting or denying other processes access to that section of the file.

Comments

The **Lock** statement locks a section of the specified file, preventing other processes from accessing that section of the file until the **Unlock** statement is issued. The **Unlock** statement unlocks a section of the specified file, allowing other processes access to that section of the file.

The **Lock** and **Unlock** statements require the following parameters:

Parameter	Description
filenumber	**Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.
record	**Long** specifying which record to lock or unlock.
start	**Long** specifying the first record within a range to be locked or unlocked.
end	**Long** specifying the last record within a range to be locked or unlocked.

For sequential files, the *record*, *start*, and *end* parameters are ignored. The entire file is locked or unlocked.

The section of the file is specified using one of the following:

Syntax	Description
No parameters	Locks or unlocks the entire file (no record specification is given).
record	Locks or unlocks the specified record number (for **Random** files) or byte (for **Binary** files).
To *end*	Locks or unlocks from the beginning of the file to the specified record (for **Random** files) or byte (for **Binary** files).
start **To** *end*	Locks or unlocks the specified range of records (for **Random** files) or bytes (for **Binary** files).

The lock range must be the same as that used to subsequently unlock the file range, and all locked ranges must be unlocked before the file is closed. Ranges within files are not unlocked automatically by BasicScript when your script terminates, which can cause file access problems for other processes. It is a good idea to group the **Lock** and **Unlock**

statements close together in the code, both for readability and so subsequent readers can see that the lock and unlock are performed on the same range. This practice also reduces errors in file locks.

Example

```
'This example creates a file named test.dat and fills it with
'ten string variable records. These are displayed in a dialog
'box. The file is then reopened for read/write, and each record
'is locked, modified, rewritten, and unlocked. The new records
'are then displayed in a dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This is record number: "
   b$ = "0"
   rec$ = ""
   message = ""
   Open "test.dat" For Random Access Write Shared As #1
   For x = 1 To 10
      rec$ = a$ & x
      Lock #1,x
      Put #1,,rec$
      Unlock #1,x
      message = message & rec$ & crlf
   Next x
   Close
   MsgBox "The records are:" & crlf & message
   message = ""
   Open "test.dat" For Random Access Read Write Shared As #1
   For x = 1 To 10
      rec$ = Mid$(rec$,1,23) & (11 - x)
      Lock #1,x
      Put #1,x,rec$
      Unlock #1,x
      message = message & rec$ & crlf
   Next x
   MsgBox "The records are: " & crlf & message
   Close
   Kill "test.dat"
End Sub
```

See Also **Open** (statement).

Platform(s) All.

Platform Notes **Macintosh:** On the Macintosh, file locking will only succeed on volumes that are shared (i.e., file sharing is on).

UNIX: Under all versions of UNIX, file locking is ignored.

Lof (function)

Syntax `Lof(`*filenumber*`)`

Description Returns a **Long** representing the number of bytes in the given file.

Comments The *filenumber* parameter is an **Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.

The file must currently be open.

Example

```
'This example creates a test file, writes ten records into it,
'then finds the length of the file and displays it in a message
'box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This is record number: "
   Open "test.dat" For Random Access Write Shared As #1
   For x = 1 To 10
      rec$ = a$ & x
      put #1,,rec$
      message = message & rec$ & crlf
   Next x
   Close
   Open "test.dat" For Random Access Read Write Shared As #1
   r% = Lof(1)
   Close
   MsgBox "The length of test.dat is: " & r%
End Sub
```

See Also **Loc** (function); **Open** (statement); **FileLen** (function).

Platform(s) All.

Log (function)

Syntax `Log(`*number*`)`

Description Returns a **Double** representing the natural logarithm of a given number.

Comments The value of *number* must be a **Double** greater than 0.

The value of *e* is 2.71828.

Example

```
'This example calculates the natural log of 100 and displays it
'in a message box.
Sub Main()
   x# = Log(100)
   MsgBox "The natural logarithm of 100 is: " & x#
```

```
End Sub
```

See Also **Exp** (function).

Platform(s) All.

Long (data type)

Syntax `Long`

Description **Long** variables are used to hold numbers (with up to ten digits of precision) within the following range:
`–2,147,483,648 <= Long <= 2,147,483,647`

Internally, longs are 4-byte values. Thus, when appearing within a structure, longs require 4 bytes of storage. When used with binary or random files, 4 bytes of storage are required.

The type-declaration character for **Long** is &.

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Integer** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CLng** (function).

Platform(s) All.

LSet (statement)

Syntax 1 `LSet` *dest* `=` *source*

Syntax 2 `LSet` *dest_variable* `=` *source_variable*

Description Left-aligns the source string in the destination string or copies one user-defined type to another.

Comments

Syntax 1

The **LSet** statement copies the source string *source* into the destination string *dest.* The *dest* parameter must be the name of either a **String** or **Variant** variable. The *source* parameter is any expression convertible to a string.

If *source* is shorter in length than *dest*, then the string is left-aligned within *dest*, and the remaining characters are padded with spaces. If *source$* is longer in length than *dest*, then *source* is truncated, copying only the leftmost number of characters that will fit in *dest*.

The *destvariable* parameter specifies a **String** or **Variant** variable. If *destvariable* is a **Variant** containing **Empty**, then no characters are copied. If *destvariable* is not convertible to a **String**, then a runtime error occurs. A runtime error results if *destvariable* is **Null**.

Syntax 2

The source structure is copied byte for byte into the destination structure. This is useful for copying structures of different types. Only the number of bytes of the smaller of the two structures is copied. Neither the source structure nor the destination structure can contain strings.

Example

```
'This example replaces a 40-character string of asterisks (*)
'with an RSet and LSet string and then displays the result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim message, tmpstr$
   tmpstr$ = String$(40, "*")
   message = "Here are two strings that have been right-" + crlf
   message = message & "and left-justified in " & _
      "a 40-character string."
   message = message & crlf & crlf
   RSet tmpstr$ = "Right->"
   message = message & tmpstr$ & crlf
   LSet tmpstr$ = "<-Left"
   message = message & tmpstr$ & crlf
   MsgBox message
End Sub
```

See Also **RSet** (statement).

Platform(s) All.

LTrim, LTrim$ (functions)

See **Trim**, **Trim$**, **LTrim**, **LTrim$**, **RTrim**, **RTrim$** (functions).

MacID (function)

Syntax `MacID(`*constant*`)`

Description Returns a value representing a collection of same-type files on the Macintosh.

Comments Since this platform does not support wildcards (i.e., * or ?), this function is the only way to specify a group of files. This function can only be used with the following statements:

```
Kill     Dir$     Shell    AppActivate
```

The *constant* parameter is a four-character string containing a file type, a resource type, an application signature, or an Apple event. A runtime error occurs if the **MacID** function is used on platforms other than the Macintosh.

Example

```
'This example retrieves the names of all the text files.
Sub Main()
   s$ = Dir$(MacID("TEXT"))'Get the first text file.
   While s$ <> ""
      MsgBox s$         'Display it.
      s$ = Dir$         'Get the next text file in the list.
   Wend
   'Delete all the text files.
   Kill MacID("TEXT")
End Sub
```

See Also **Kill** (statement); **Dir**, **Dir$** (functions); **Shell** (function); **AppActivate** (statement).

Platform(s) Macintosh.

MacScript (statement)

Syntax `MacScript` *script*

Description Executes the specified AppleScript script.

Comments When using the MacScript statement, you can separate multiple lines by embedding carriage returns:

```
MacScript "Beep" + Chr(13) + "Display Dialog ""Hello"""
```

If embedding carriage returns proves cumbersome, you can use the **Inline** statement. The following **Inline** statement is equivalent to the above example:

```
Inline MacScript
   Beep
   Display Dialog "Hello"
End Inline
```

Example

```
Sub Main()
```

```
   MacScript "display dialog ""AppleScript"""
End Sub
```

See Also **Inline** (statement).

Platform(s) Macintosh..

Main (statement)

Syntax
```
Sub Main()
End Sub
```

Description Defines the subroutine where execution begins.

Example
```
Sub Main()
   MsgBox "This is the Main() subroutine and entry point."
End Sub
```

Platform(s) All.

Mci (function)

Syntax `Mci(`*command$*`,`*result$* `[,`*error$*`])`

Description Executes an **Mci** command, returning an **Integer** indicating whether the command was successful.

Comments The **Mci** function takes the following parameters:

Parameter	Description
command$	**String** containing the command to be executed.
result$	**String** variable into which the result is placed. If the command doesn't return anything, then a zero-length string is returned. To ignore the returned string, pass a zero-length string: `r% = Mci("open chimes.wav type waveaudio","")`
error$	Optional **String** variable into which an error string will be placed. A zero-length string will be returned if the function is successful.

The **Mci** function returns 0 if successul. Otherwise, an non-zero **Integer** is returned indicating the error.

Examples
```
'This first example plays a wave file. The wave file is played
'to completion before execution can continue.
Sub Main()
```

```
   Dim result As String
   Dim ErrorMessage As String
   Dim Filename As String
   Dim rc As Integer
   'Establish name of file in the Windows directory.
   Filename = FileParse$(System.WindowsDirectory$ + "\" _
      + "chimes.wav")
   'Open the file and driver.
   rc = Mci("open " & Filename & _
      " type waveaudio alias CoolSound","",ErrorMessage)
   If (rc) Then
      'Error occurred--display error message to user.
      MsgBox ErrorMessage
      Exit Sub
   End If
   rc = Mci("play CoolSound wait","","")  'Wait for sound to
                                          'finish.
   rc = Mci("close CoolSound","","")      'Close driver and file.
End Sub

'This next example shows how to query an Mci device and play an
'MIDI file in the background.
Sub Main()
   Dim result As String
   Dim ErrMsg As String
   Dim Filename As String
   Dim rc As Integer
   'Check to see whether MIDI device can play for us.
   rc = Mci("capability sequencer can play",result,ErrorMessage)
   'Check for error.
   If rc Then
      MsgBox ErrorMessage
      Exit Sub
   End If
   'Can it play?
   If result <> "true" Then
      MsgBox "MIDI device is not capable of playing."
      Exit Sub
   End If
   'Assemble a filename from the Windows directory.
   Filename = FileParse$(System.WindowsDirectory$ & "\" _
      & "canyon.mid")
   'Open the driver and file.
   rc = Mci("open " & Filename & _
      " type sequencer alias song",result$,ErrMsg)
   If rc Then
      MsgBox ErrMsg
      Exit Sub
```

```
   End If
   rc = Mci("play song","","")'Play in the background.
   MsgBox "Press OK to stop the music.",ebOKOnly
   rc = Mci("close song","","")
End Sub
```

See Also **Beep** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows:** The **Mci** function accepts any **Mci** command as defined in the *Multimedia Programmers Reference* in the Windows 3.1 SDK.

Menu (statement)

Syntax `Menu` *MenuItem$*

Description Issues the specified menu command from the active window of the active application.

Comments The *MenuItem$* parameter specifies the complete menu item name, with each menu level being separated by a period. For example, the "Open" command on the "File" menu is represented by "File.Open". Cascading menu items may have multiple periods, one for each pop-up menu, such as "File.Layout.Vertical". Menu items can also be specified using numeric index values. For example, to select the third menu item from the File menu, use "File.#3". To select the fourth item from the third menu, use "#3.#4".

Items from an application's system menu can be selected by beginning the menu item specification with a period, such as ".Restore" or ".Minimize".

A runtime error will result if the menu item specification does not specify a menu item. For example, "File" specifies a menu pop-up rather than a menu item, and "File.Blah Blah" is not a valid menu item.

When comparing menu item names, this statement removes periods (.), spaces, and the ampersand. Furthermore, all characters after a backspace or tab are removed. Thus, the menu item "&Open...\aCtrl+F12" translates simply to "Open".

A runtime error is generated if the menu item cannot be found or is not enabled at the time that this statement is encountered.

Examples

```
Sub Main()
   Menu "File.Open"
   Menu "Format.Character.Bold"
   Menu ".Restore"'Command from system menu
   Menu "File.#2"
End Sub
```

See Also **MenuItemChecked** (function); **MenuItemEnabled** (function); **MenuItemExists** (function).

Platform(s) Windows.

MenuItemChecked (function)

Syntax `MenuItemChecked(`*MenuItemName$*`)`

Description Returns **True** if the given menu item exists and is checked; returns **False** otherwise.

Comments The *MenuItemName$* parameter specifies a complete menu item or menu item pop-up following the same format as that used by the **Menu** statement.

Example
```
'This example turns the ruler off if it is on.
Sub Main()
   If MenuItemChecked("View.Ruler") Then Menu "View.Ruler"
End Sub
```

See Also **Menu** (statement); **MenuItemEnabled** (function); **MenuItemExists** (function).

Platform(s) Windows.

MenuItemEnabled (function)

Syntax `MenuItemEnabled(`*MenuItemName$*`)`

Description Returns **True** if the given menu item exists and is enabled; returns **False** otherwise.

Comments The *MenuItemName$* parameter specifies a complete menu item or menu item pop-up following the same format as that used by the **Menu** statement.

Example
```
'This example only pastes if there is something in the Clipboard.
Sub Main()
   If MenuItemEnabled("Edit.Paste") Then
      Menu "Edit.Paste"
   Else
      MsgBox "There is nothing in the Clipboard.",ebOKOnly
   End If
End Sub
```

See Also **Menu** (statement); **MenuItemChecked** (function); **MenuItemExists** (function).

Platform(s) Windows.

MenuItemExists (function)

Syntax `MenuItemExists(`*MenuItemName$*`)`

Description Returns **True** if the given menu item exists; returns **False** otherwise.

Comments The *MenuItemName$* parameter specifies a complete menu item or menu item pop-up following the same format as that used by the **Menu** statement.

Examples
```
Sub Main()
   If MenuItemExists("File.Open") Then Beep
   If MenuItemExists("File") Then MsgBox "There is a File menu."
End Sub
```

See Also **Menu** (statement); **MenuItemChecked** (function); **MenuItemEnabled** (function).

Platform(s) Windows.

Mid, Mid$, MidB, MidB$ (functions)

Syntax
```
Mid[$](string, start [,length])
MidB[$](string, start [,length])
```

Description Returns a substring of the specified string, beginning with *start*, for *length* characters (for **Mid** and **Mid$**) or bytes (for **MidB** and **MidB$**).

Comments The **Mid** and **Mid$** functions return a substring starting at character position *start* and will be *length* characters long. The **MidB** and **MidB** functions return a substring starting at byte position *start* and will be *length* bytes long.

The **Mid$** and **MidB$** functions return a **String**, whereas the **Mid** and **MidB** functions return a **String** variant.

These functions take the following named parameters:

Named Parameter	Description
string	Any **String** expression containing the text from which data are returned.
start	**Integer** specifying the position where the substring begins. If *start* is greater than the length of *string*, then a zero-length string is returned.
length	**Integer** specifying the number of characters or bytes to return. If this parameter is omitted, then the entire string is returned, starting at *start*.

The **Mid** function will return **Null** if *string* is **Null**.

The **MidB** and **MidB$** functions are used to return a substring of bytes from a string containing byte data.

Example
```
'This example displays a substring from the middle of a string
'variable using the Mid$ function and replaces the first four
```

```
'characters with "NEW " using the Mid$ statement.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This is the Main string containing text."
   b$ = Mid$(a$,13,Len(a$))
   Mid$ (b$,1) = NEW "
   MsgBox a$ & crlf & b$
End Sub
```

See Also **InStr, InStrB** (functions); **Option Compare** (statement); **Mid**, **Mid$, Mid**, **Mid$** (statements).

Platform(s) All.

Mid, Mid$, MidB, MidB$ (statements)

Syntax

```
Mid[$](variable,start[,length]) = newvalue
MidB[$](variable,start[,length]) = newvalue
```

Description Replaces one part of a string with another.

Comments The **Mid**/**Mid$** statements take the following parameters:

Parameter	Description
variable	**String** or **Variant** variable to be changed.
start	**Integer** specifying the character position (for **Mid** and **Mid$**) or byte position (for **MidB** and **MidB$**) within *variable* where replacement begins. If *start* is greater than the length of *variable*, then *variable* remains unchanged.
length	**Integer** specifying the number of characters or bytes to change. If this parameter is omitted, then the entire string is changed, starting at *start*.
newvalue	Expression used as the replacement. This expression must be convertible to a **String**.

The resultant string is never longer than the original length of *variable*.

With **Mid** and **MidB**, *variable* must be a **Variant** variable convertible to a **String**, and *newvalue* is any expression convertible to a string. A runtime error is generated if either variant is **Null**.

The **MidB** and **MidB$** statements are used to replace a substring of bytes, whereas **Mid** and **Mid$** are used to replace a substring of characters.

Example

```
'This example displays a substring from the middle of a string
'variable using the Mid$ function, replacing the first four
'characters with "NEW " using the Mid$ statement.
```

```
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This is the Main string containing text."
   b$ = Mid$(a$,13,Len(a$))
   Mid$(b$,1) = "NEW "
   MsgBox a$ & crlf & b$
End Sub
```

See Also **Mid**, **Mid$, Mid**B, **MidB$** (functions); **Option Compare** (statement).

Platform(s) All.

Minute (function)

Syntax `Minute(`*time*`)`

Description Returns the minute of the day encoded in the specified *time* parameter.

Comments The value returned is as an **Integer** between 0 and 59 inclusive.

The *time* parameter is any expression that converts to a **Date**.

Example

```
'This example takes the current time; extracts the hour, minute,
'and second; and displays them as the current time.
Sub Main()
   xt# = TimeValue(Time$())
   xh# = Hour(xt#)
   xm# = Minute(xt#)
   xs# = Second(xt#)
   MsgBox "The current time is: " & xh# & ":" & xm# & ":" & xs#
End Sub
```

See Also **Day** (function); **Second** (function); **Month** (function); **Year** (function); **Hour** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

MIRR (function)

Syntax `MIRR(`*valuearray*`(),`*financerate*`,`*reinvestrate*`)`

Description Returns a **Double** representing the modified internal rate of return for a series of periodic payments and receipts.

Comments The modified internal rate of return is the equivalent rate of return on an investment in which payments and receipts are financed at different rates. The interest cost of investment and the rate of interest received on the returns on investment are both factors in the calculations.

The **MIRR** function requires the following named parameters:

Named Parameter	Description
valuearray()	Array of **Double** numbers representing the payments and receipts. Positive values are payments (invested capital), and negative values are receipts (returns on investment). There must be at least one positive (investment) value and one negative (return) value.
financerate	**Double** representing the interest rate paid on invested monies (paid out).
reinvestrate	**Double** representing the rate of interest received on incomes from the investment (receipts).

The *financerate* and *reinvestrate* parameters should be expressed as percentages. For example, 11 percent should be expressed as 0.11.

To return the correct value, be sure to order your payments and receipts in the correct sequence.

Example

```
'This example illustrates the purchase of a lemonade stand for
'$800 financed with money borrowed at 10%. The returns are
'estimated to accelerate as the stand gains popularity. The
'proceeds are placed in a bank at 9 percent interest. The
'incomes are estimated (generated) over 12 months. This program
'first generates the income stream array in two For...Next
'loops, and then the modified internal rate of return is
'calculated and displayed. Notice that the annual rates are
'normalized to monthly rates by dividing them by 12.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim valu#(12)
   valu(1) = -800            'Initial investment
   message = valu(1) & ", "
   For x = 2 To 5
      valu(x) = 100 + (x * 2)'Incomes months 2-5
      message = message & valu(x) & ", "
   Next x
   For x = 6 To 12
      valu(x) = 100 + (x * 10)   'Incomes months 6-12
      message = message & valu(x) & ", "
   Next x
   retrn# = MIRR(valu,.1/12,.09/12)    'Note: normalized annual
                                       'rates
   message = "The values: " & crlf & message & crlf & crlf
   MsgBox message & "Modified rate: " & Format(retrn#,"Percent")
End Sub
```

See Also **Fv** (function); **IRR** (function); **Npv** (function); **Pv** (function).

Platform(s) All.

MkDir (statement)

Syntax `MkDir` *path*

Description Creates a new directory as specified by *path*.

Example

```
'This example creates a new directory on the default drive. If
'this causes an error, then the error is displayed and the
'program terminates. If no error is generated, the directory is
'removed with the RmDir statement.
Sub Main()
   On Error Resume Next
   MkDir "TestDir"
   If Err <> 0 Then
      MsgBox "The following error occurred: " & Error(Err)
   Else
      MsgBox "Directory was created and is about to be removed."
      RmDir "TestDir"
   End If
End Sub
```

See Also **ChDir** (statement); **ChDrive** (statement); **CurDir**, **CurDir$** (functions); **Dir**, **Dir$** (functions); **RmDir** (statement).

Platform(s) All.

Platform Notes **Windows:** This command behaves the same as the DOS "mkdir" command.

Mod (operator)

Syntax *expression1* `Mod` *expression2*

Description Returns the remainder of *expression1* / *expression2* as a whole number.

Comments If both expressions are integers, then the result is an integer. Otherwise, each expression is converted to a **Long** before performing the operation, returning a **Long**.

A runtime error occurs if the result overflows the range of a **Long**.

If either expression is **Null**, then **Null** is returned. **Empty** is treated as 0.

Example

```
'This example uses the Mod operator to determine the value of a
'randomly selected card where card 1 is the ace (1) of clubs
'and card 52 is the king (13) of spades. Since the values recur
```

```
'in a sequence of 13 cards within 4 suits, we can use the Mod
'function to determine the value of any given card number.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   cval$ = "ACE,TWO,THREE,FOUR,FIVE,"
   cval$ = cval$ + "SIX,SEVEN,EIGHT,NINE,TEN,JACK,QUEEN,KING"
   Randomize
   card% = Random(1,52)
   value = card% Mod 13
   If value = 0 Then value = 13
   CardNum$ = Item$(cval,value)
   If card% < 53 Then suit$ = "spades"
   If card% < 40 Then suit$ = "hearts"
   If card% < 27 Then suit$ = "diamonds"
   If card% < 14 Then suit$ = "clubs"
   message = "Card number " & card% & " is the "
   message = message & CardNum & " of " & suit$
   MsgBox message
End Sub
```

See Also / (operator); \ (operator).

Platform(s) All.

Month (function)

Syntax `Month(`*date*`)`

Description Returns the month of the date encoded in the specified *date* parameter.

Comments The value returned is as an **Integer** between 1 and 12 inclusive.

The *date* parameter is any expression that converts to a **Date**.

Example

```
'This example returns the current month in a dialog box.
Sub Main()
   mons$ = "Jan., Feb., Mar., Apr., May, Jun., "
   mons$ = "Jul., Aug., Sep., Oct., Nov., Dec."
   tdate$ = Date$
   tmonth! = Month(DateValue(tdate$))
   MsgBox "The current month is: " & Item$(mons$,tmonth!)
End Sub
```

See Also **Day** (function); **Minute** (function); **Second** (function); **Year** (function); **Hour** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

Msg.Close (method)

Syntax `Msg.Close`

Description Closes the modeless message dialog box.

Comments Nothing will happen if there is no open message dialog box.

Example

```
Sub Main()
   Msg.Open "Printing. Please wait...",0,True,True
   Sleep 3000
   Msg.Close
End Sub
```

See Also **Msg.Open** (method); **Msg.Thermometer** (property); **Msg.Text** (property).

Platform(s) Windows, Win32.

Msg.Open (method)

Syntax `Msg.Open` *prompt*`,`*timeout*`,`*cancel*`,`*thermometer* `[,`*XPos*`,`*YPos*`]`

Description Displays a message in a dialog box with an optional Cancel button and thermometer.

Comments The **Msg.Open** method takes the following named parameters:

Parameter	Description
prompt	**String** containing the text to be displayed. The text can be changed using the **Msg.Text** property.
timeout	**Integer** specifying the number of seconds before the dialog box is automatically removed. The *timeout* parameter has no effect if its value is 0.
cancel	**Boolean** controlling whether or not a Cancel button appears within the dialog box beneath the displayed message. If this parameter is **True**, then a Cancel button appears. If it is not specified or **False**, then no Cancel button is created. If a user chooses the Cancel button at runtime, a trappable runtime error is generated (error number 18). In this manner, a message dialog box can be displayed and processing can continue as normal, aborting only when the user cancels the process by choosing the Cancel button.

Parameter	Description
thermometer	**Boolean** controlling whether the dialog box contains a thermometer. If this parameter is **True**, then a thermometer is created between the text and the optional Cancel button. The thermometer initially indicates 0% complete and can be changed using the **Msg.Thermometer** property.
XPos, YPos	**Integer** coordinates specifying the location of the upper left corner of the message box, in twips (twentieths of a point). If these parameters are not specified, then the window is centered on top of the application.

Unlike other dialog boxes, a message dialog box remains open until the user selects Cancel, the timeout has expired, or the **Msg.Close** method is executed (this is sometimes referred to as modeless).

Only a single message window can be opened at any one time. The message window is removed automatically when a script terminates.

The Cancel button, if present, can be selected using either the mouse or keyboard. However, these events will never reach the message dialog unless you periodically call **DoEvents** from within your script.

Example

```
'This example displays several types of message boxes.
Sub Main()
   Msg.Open "Printing. Please wait...",0,True,False
   Sleep 3000
   Msg.Close
   Msg.Open "Printing. Please wait...",0,True,True
   For x = 1 to 100
      Msg.Thermometer = x
   Next x
   Sleep 1000
   Msg.Close
End Sub
```

See Also **Msg.Close** (method); **Msg.Thermometer** (property); **Msg.Text** (property).

Platform(s) Windows, Win32.

Msg.Text (property)

Syntax `Msg.Text [= newtext$]`

Description Changes the text within an open message dialog box (one that was previously opened with the **Msg.Open** method).

Comments The message dialog box is not resized to accommodate the new text.

A runtime error will result if a message dialog box is not currently open (using **Msg.Open**).

Example

```
'This example creates a modeless message box, leaving room in
'the message text for the record number. This box contains a
'Cancel button.
Sub Main()
   Msg.Open "Reading Record",0,True,False
   For i = 1 To 100
      'Read a record here.
      'Update the modeless message box.
      Sleep 100
      Msg.Text ="Reading record " & i
   Next i
   Msg.Close
End Sub
```

See Also **Msg.Close** (method); **Msg.Open** (method); **Msg.Thermometer** (property).

Platform(s) Windows, Win32.

Msg.Thermometer (property)

Syntax `Msg.Thermometer [= percentage]`

Description Changes the percentage filled indicated within the thermometer of a message dialog box (one that was previously opened with the **Msg.Open** method).

Comments A runtime error will result if a message box is not currently open (using **Msg.Open**) or if the value of *percentage* is not between 0 and 100 inclusive.

Example

```
'This example create a modeless message box with a thermometer
'and a Cancel button. This example also shows how to process the
'clicking of the Cancel button.
Sub Main()
   On Error Goto ErrorTrap
   Msg.Open "Reading records from file...",0,True,True
   For i = 1 To 100
      'Read a record here.
      'Update the modeless message box.
      Msg.Thermometer =i
      DoEvents
      Sleep 50
   Next i
   Msg.Close
   On Error Goto 0'Turn error trap off.
   Exit Sub
ErrorTrap:
```

```
   If Err = 809 Then
      MsgBox "Cancel was pressed!"
      Exit Sub    'Reset error handler.
   End If
End Sub
```

See Also **Msg.Close** (method); **Msg.Open** (method); **Msg.Text** (property).

Platform(s) Windows, Win32.

MsgBox (function)

Syntax `MsgBox(`*prompt* `[, [`*buttons*`] [,[`*title*`] [,`*helpfile*`,`*context*`]]])`

Description Displays a message in a dialog box with a set of predefined buttons, returning an **Integer** representing which button was selected.

Comments The **MsgBox** function takes the following named parameters:

Named Parameter	Description
prompt	Message to be displayed—any expression convertible to a **String**. End-of-lines can be used to separate lines (either a carriage return, line feed, or both). If a given line is too long, it will be word-wrapped. If *prompt* contains character 0, then only the characters up to the character 0 will be displayed. The width and height of the dialog box are sized to hold the entire contents of *prompt*. A runtime error is generated if *prompt* is **Null**.
buttons	**Integer** specifying the type of dialog box (see below).
title	Caption of the dialog box. This parameter is any expression convertible to a **String**. If it is omitted, then "BasicScript" is used. A runtime error is generated if *title* is **Null**.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

The **MsgBox** function returns one of the following values:

Constant	Value	Description
ebOK	1	OK was pressed.
ebCancel	2	Cancel was pressed.
ebAbort	3	Abort was pressed.
ebRetry	4	Retry was pressed.
ebIgnore	5	Ignore was pressed.
ebYes	6	Yes was pressed.
ebNo	7	No was pressed.

The *buttons* parameter is the sum of any of the following values:

Constant	Value	Description
ebOKOnly	0	Displays OK button only.
ebOKCancel	1	Displays OK and Cancel buttons.
ebAbortRetryIgnore	2	Displays Abort, Retry, and Ignore buttons.
ebYesNoCancel	3	Displays Yes, No, and Cancel buttons.
ebYesNo	4	Displays Yes and No buttons.
ebRetryCancel	5	Displays Retry and Cancel buttons.
ebCritical	16	Displays "stop" icon.
ebQuestion	32	Displays "question mark" icon.
ebExclamation	48	Displays "exclamation point" icon.
ebInformation	64	Displays "information" icon.
ebDefaultButton1	0	First button is the default button.
ebDefaultButton2	256	Second button is the default button.
ebDefaultButton3	512	Third button is the default button.
ebApplicationModal	0	Application modal—the current application is suspended until the dialog box is closed.
ebSystemModal	4096	System modal—all applications are suspended until the dialog box is closed.

The default value for *buttons* is 0 (display only the OK button, making it the default).

If both the *helpfile* and *context* parameters are specified, then context-sensitive help can be invoked using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Breaking Text across Lines

The *prompt* parameter can contain end-of-line characters, forcing the text that follows to start on a new line. The following example shows how to display a string on two lines:

```
MsgBox "This is on" + Chr(13) + Chr(10) + "two lines."
```

The carriage-return or line-feed characters can be used by themselves to designate an end-of-line.

Example

```
Sub Main
   MsgBox "This is a simple message box."
   MsgBox "This is a message box with a title and an icon.", _
      ebExclamation,"Simple"
   MsgBox "This message box has OK and Cancel buttons.", _
      ebOkCancel
   MsgBox "This message box has Yes, No, and Cancel buttons.", _
      ebYesNoCancel Or ebDefaultButton2
   MsgBox "This message box has Yes and No buttons.", _
      ebYesNo
   MsgBox "This message box has Retry and Cancel buttons.", _
      ebRetryCancel
   MsgBox "This message box is system modal!",ebSystemModal
End Sub
```

See Also **AskBox, AskBox$** (functions); **AskPassword**, **AskPassword$** (functions); **InputBox**, **InputBox$** (functions); **OpenFileName$** (function); **SaveFileName$** (function); **SelectBox** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes The appearance of the **MsgBox** dialog box and its icons differs slightly depending on the platform.

MsgBox (statement)

Syntax `MsgBox` *prompt* `[, [`*buttons*`] [,[`*title*`] [,` *helpfile*`,` *context*`]]]`

Description This command is the same as the **MsgBox** function, except that the statement form does not return a value. See **MsgBox** (function).

Example

```
Sub Main()
   MsgBox "This is text displayed in a message box."  'Display
                                                      'text.
   MsgBox "The result is: " & (10 * 45)'Display a number.
```

```
End Sub
```

See Also **AskBox, AskBox$** (functions); **AskPassword**, **AskPassword$** (functions); **InputBox**, **InputBox$** (functions); **OpenFileName$** (function); **SaveFileName$** (function); **SelectBox** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Name (statement)

Syntax `Name` *oldfile$* `As` *newfile$*

Description Renames a file.

Comments Each parameter must specify a single filename. Wildcard characters such as * and ? are not allowed.

Some platforms allow naming of files to different directories on the same physical disk volume. For example, the following rename will work under Windows:

```
Name "c:\samples\mydoc.txt" As "c:\backup\doc\mydoc.bak"
```

You cannot rename files across physical disk volumes. For example, the following will error under Windows:

```
Name "c:\samples\mydoc.txt" As "a:\mydoc.bak"   'This will
                                                ’error!
```

To rename a file to a different physical disk, you must first copy the file, then erase the original:

```
FileCopy "c:\samples\mydoc.txt","a:\mydoc.bak" 'Make a copy.
Kill "c:\samples\mydoc.txt"   'Delete the original.
```

Example

```
’This example creates a file called test.dat and then renames it
’to test2.dat.
Sub Main()
   On Error Resume Next
   If FileExists("test.dat") Then
      Name "test.dat" As "test2.dat"
      If Err <> 0 Then
         message = "File exists and cannot be renamed! Error: " _
            & Err
      Else
         message = "File exists and renamed to test2.dat."
      End If
   Else
      Open "test.dat" For Output As #1
      Close
      Name "test.dat" As "test2.dat"
      If Err <> 0 Then
         message = "File created but not renamed! Error: " & Err
      Else
         message = "File created and renamed to test2.dat."
      End If
   End If
   MsgBox message
End Sub
```

See Also **Kill** (statement); **FileCopy** (statement).

Platform(s) All.

Named Parameters (topic)

Many language elements in BasicScript support named parameters. Named parameters allow you to specify parameters to a function or subroutine by name rather than in adherence to a predetermined order. The following table contains examples showing various calls to **MsgBox** both using parameter by both name and position.

By Name	**MsgBox Prompt:= "Hello, world."**
By Position	**MsgBox "Hello, world."**
By Name	**MsgBox Title:="Title", Prompt:="Hello, world."**
By Position	**MsgBox "Hello, world",,"Title"**
By Name	**MsgBox HelpFile:="BASIC.HLP", _ Prompt:="Hello, world.", Context:=10**
By Position	**MsgBox "Hello, world.",,,"BASIC.HLP",10**

Using named parameter makes your code easier to read, while at the same time removes you from knowing the order of parameter. With function that require many parameters, most of which are optional (such as **MsgBox**), code becomes significantly easier to write and maintain.

When supported, the names of the named parameter appear in the description of that language element.

When using named parameter, you must observe the following rules:

- Named parameter must use the parameter name as specified in the description of that language element. Unrecognized parameter names cause compiler errors.
- All parameters, whether named or positional, are separated by commas.
- The parameter name and its associated value are separated with **:=**
- If one parameter is named, then all subsequent parameter must also be named as shown below:

```
MsgBox "Hello, world", Title:="Title"     'OK
MsgBox Prompt:="Hello, world.",,"Title"   'WRONG!!!
```

Net.AddCon (method)

Syntax `Net.AddCon` *netpath$*`,[`*password$*`],[`*localname$*`] [,[`*username$*`] [,`*permanent*`]]`

Description Redirects a local device (a disk drive or printer queue) to the specified shared device or remote server.

Comments The **Net.AddCon** method takes the following parameters:

Parameter	Description
netpath$	**String** containing the name of the shared device or the name of a remote server. This parameter can contain the name of a shared printer queue (such as that returned by **Net.Browse[1]**) or the name of a network path (such as that returned by **Net.Browse[0]**).
password$	**String** containing the password for the given device or server. This parameter is mainly used to specify the password on a remote server. If *password$* is not specified, then the default password is used.
localname$	**String** containing the name of the local device being redirected, such as "LPT1" or "D:". If *localname$* is not specified, then a connection is made to the network resource without redirecting a local device.
username$	Specifies the name of the user making the connection.
permanent	Specifies if the connection should be restored during subsequent logon operations. Only a successful connection will persist in this manner. Connections are assumed to be permanent if this parameter is omitted. Connections established when *localname$* is missing are never permanent.

A runtime error will result if no network is present.

Example

```
'This example sets N: so that it refers to the network path
'SYS:\PUBLIC.
Sub Main()
   Net.AddCon "SYS:\PUBLIC","","N:"
End Sub
```

See Also **Net.CancelCon** (method); **Net.GetCon$** (method).

Platform(s) Windows, Win32.

Platform Notes **Windows:** On Windows platforms, the *localname$* parameter cannot be omitted. The *username$* and *permanent* parameters are ignored.

Win32: On Win32 platforms, if *username$* is omitted, then the default user for the current process is used. The permanent parameter is always **True** under Win32s.

Net.Browse$ (method)

Syntax `Net.Browse$(`*type*`)`

Description Calls the currently installed network's browse dialog box, requesting a particular type of information.

Comments The *type* parameter is an **Integer** specifying the type of dialog box to display:

Type	Description
0	Displays a dialog box that allows the user to browse network volumes and directories. Choosing OK returns the completed pathname as a **String**.
1	Displays a dialog box that allows the user to browse the network's printer queues. Choosing OK returns the complete name of that printer queue as a **String**. This string is the same format as required by the **Net.AddCon** method.
2	Displays the disconnect dialog for disk resources.
3	Displays the disconnect dialog for printer resources.

This dialog box differs depending on the type of network installed.

A runtime error will result if no network is present.

Example

```
'This example retrieves a valid network path.
Sub Main()
   s$ = Net.Browse$(0)
   If s$ <> "" Then
      MsgBox "The following network path was selected: " & s$
   Else
      MsgBox "Dialog box was canceled."
   End If
End Sub
```

See Also **Net.Dialog** (method).

Platform(s) Windows, Win32.

Platform Notes **Windows:** Under Windows, types 2 and 3 are not supported.

Win32: On Win32 platforms, this method always returns an empty string. Instead, each dialog automatically establishes the connection.

Types 1 and 3 are only supported under Windows 95 and Windows NT version 4.0 or later.

Net.CancelCon (method)

Syntax `Net.CancelCon` *connection$* `[,[`*isForce*`] [,isPermanent]]`

Description Cancels a network connection.

Comments The Net.**CancelCon** method takes the following parameters:

Parameter	Description
connection$	String containing the name of the device to cancel, such as "LPT1" or "D:". If *connection$* specifies a local device, then only that local device is disconnected. If *connection$* specifies a remote device, then all local devices attached to that remote device are disconnected.
isForce	**Boolean** specifying whether to force the cancellation of the connection if there are open files or open print jobs. If this parameter is **True**, then this method will close all open files and open print jobs before the connection is closed. If this parameter is **False**, this the method will issue a runtime error if there are any open files or open print jobs. If omitted, then *isForce* is assumed to be **True**.
isPermanent	**Boolean** specifying whether the disconnection should be temporary or should persist to subsequent logon operations. If this parameter is missing, then it is assumed to be **True**.

A runtime error will result if no network is present.

Example

```
'This example deletes the drive mapping associated with drive N:.
Sub Main()
   Net.CancelCon "N:"
End Sub
```

See Also **Net.AddCon** (method); **Net.GetCon$** (method).

Platform(s) Windows, Win32.

Platform Notes **Windows:** Under Windows, *isPermanent* is ignored.

Win32: The Net.CancelCon requires Win32s version 1.3 or later.

Net.Dialog (method)

Syntax `Net.Dialog`

Description Displays the dialog box that allows configuration of the currently installed network.

Comments The displayed dialog box depends on the currently installed network. The dialog box is modal—script execution will be paused until the dialog box is completed.

A runtime error will result if no network is present.

Example

```
'This example invokes the network driver dialog box.
Sub Main()
   Net.Dialog
End Sub
```

See Also **Net.Browse$** (method).

Platform(s) Windows.

Net.GetCaps (method)

Syntax `Net.GetCaps(`*type* `[,`*localname$*`])`

Description Returns an **Integer** specifying information about the network and its capabilities.

Comments The **Net.GetCaps** method takes the following parameters:

Parameter	Description
type	An **Integer** specifying what type of information to retrieve. This parameter is different from platform to platform.
localname$	A **String** specifying the name of the local device to which is attached to the network device to be queried. If this parameter is missing, then information about the first network device is returned.

A runtime error will result if no network is present.

Examples

```
Sub Main()
   'This example checks the type of network.
   If Net.GetCaps(2) = 768 Then MsgBox "This is a Novell network."
   'Check whether the net supports retrieval of the user name.
   If Net.GetCaps(4) And 1 Then MsgBox "User name is: " +
Net.User$
   'This checks whether this net supports the Browse dialog
boxes.
   If Net.GetCaps(6) And &H0010 Then MsgBox Net.Browse$(1)
End Sub
```

Platform(s) Windows, Win32.

Platform Notes **Windows:** Under Windows, since only one network connection is possible at any given time, the **localname$** parameter is ignored.

The *type* parameter for Win16 platforms can be any of the values described in the following table:

Value of *type*	Description
1	Returns the version of the driver specification to which the currently installed network driver conforms. The high byte of the returned value contains the major version number and the low byte contains the minor version number. These values can be retrieved using the following code: *MajorVersionNumber* = `Net.GetCaps(1) \ 256` *MinorVersionNumber* = `Net.GetCaps(1) And &H00FF`
2	Returns the type of network. The network type is returned in the high byte and the subnetwork type is returned in the low byte. These values can be obtained using the following code: *NetType* = `Net.GetCaps(2) \ 256` *SubNetType* = `Net.GetCaps(2) And &H00FF` Using the above values, *NetType* can be any of the following values: 0 No network is installed. 1 Microsoft Network. 2 Microsoft LAN Manager. 3 Novell NetWare. 4 Banyan Vines. 5 10Net. 6 Locus 7 SunSoft PC NFS. 8 LanStep. 9 9 Tiles. 10 Articom Lantastic. 11 IBM AS/400. 12 FTP Software FTP NFS. 13 DEC Pathworks.

Value of *type*	Description
	If *NetType* is 128, then *SubNetType* is any of the following values (you can test for any of these values using the **And** operator): bit &H0001 Microsoft Network. bit &H0002 Microsoft LAN Manager. bit &H0004 Windows for Workgroups. bit &H0008 Novell NetWare. bit &H0010 Banyan Vines. bit &H0080 Other unspecified network.
3	Returns the network driver version number.
4	Returns 1 if the **Net.User$** property is supported; returns 0 otherwise.
6	Returns any of the following values indicating which connections are supported (you can test for these values using the **And** operator): bit &H0001 Driver supports **Net.AddCon**. bit &H0002 Driver supports **Net.CancelCon**. bit &H0004 Driver supports **Net.GetCon**. bit &H0008 Driver supports auto connect. bit &H0010 Driver supports **Net.Browse$**.

Value of *type*	Description
7	Returns a value indicating which printer functions are available (you can test for these values using the **And** operator):
	bit &H0002 Driver supports open print job.
	bit &H0004 Driver supports close print job.
	bit &H0010 Driver supports hold print job.
	bit &H0020 Driver supports release print job.
	bit &H0040 Driver supports cancel print job.
	bit &H0080 Driver supports setting the number of print copies.
	bit &H0100 Driver supports watch print queue.
	bit &H0200 Driver supports unwatch print queue.
	bit &H0400 Driver supports locking queue data.
	bit &H0800 Driver supports unlocking queue data.
	bit &H1000 Driver supports queue change message.
	bit &H2000 Driver supports abort print job.
	bit &H4000 Driver supports no arbitrary lock.
	bit &H8000 Driver supports write print job.
8	Returns a value indicating which dialog functions are available (you can test for these values using the **And** operator):
	bit &H0001 Driver supports Device Mode dialog.
	bit &H0002 Driver supports the Browse dialog.
	bit &H0004 Driver supports the Connect dialog.
	bit &H0008 Driver supports the Disconnect dialog.
	bit &H0010 Driver supports the View Queue dialog.
	bit &H0020 Driver supports the Property dialog.
	bit &H0040 Driver supports the Connection dialog.
	bit &H0080 Driver supports the Printer Connect dialog.
	bit &H0100 Driver supports the Shares dialog.
	bit &H0200 Driver supports the Share As dialog.

Win32: For Win32 platforms, the type parameter can be any of the following values:

Value of *type*	Description	
1	Always returns 0	
2	Network type:	
	0	No network is installed.
	1	Microsoft Network.
	2	Microsoft LAN Manager.
	3	Novell NetWare.
	4	Banyan Vines.
	5	10Net.
	6	Locus
	7	SunSoft PC NFS.
	8	LanStep.
	9	9 Titles.
	10	Articom Lantastic.
	11	IBM AS/400.
	12	FTP Software FTP NFS.
	13	DEC Pathworks.
3	Version of the network with the major version in the high byte and the minor version in the low byte: *Major* `= Net.GetCaps(2) \ 256` *Minor* `= Net.GetCaps(2) And &H00FF`	

Net.GetCon$ (method)

Syntax `Net.GetCon$(`*localname$*`)`

Description Returns the name of the network resource associated with the specified redirected local device.

Comments The *localname$* parameter specifies the name of the local device, such as "LPT1" or "D:".

The function returns a zero-length string if the specified local device is not redirected.

A runtime error will result if no network is present.

Example

```
'This example finds out where drive Z is mapped.
Sub Main()
```

```
   NetPath$ = Net.GetCon$("Z:")
   MsgBox "Drive Z is mapped as " & NetPath$
End Sub
```

See Also **Net.CancelCon** (method); **Net.AddCon** (method).

Platform(s) Windows Win32.

Net.User$ (method)

Syntax `Net.User$ [([`*localname$*`])]`

Description Returns the name of the user on the network.

Comments The *localname$* parameter is a **String** specifying the name of the local device that the user has made a connection to. If this parameter is omitted, then the name of the user of the current process is used.

If **localname$** is the name of a network device and the user is connected to that resource using different names, then the network provider may not be able to resolve which user name to return. In this case, the provider may make an arbitrary choice from the possible user names.

A runtime error is generated if the network is not installed.

Examples

```
Sub Main()
   'This example tells the user who he or she is.
   MsgBox "You are " & Net.User$
   'This example makes sure this capability is supported.
   If Net.GetCaps(4) And 1 Then MsgBox "You are " & Net.User$
End Sub
```

Platform(s) Windows, Win32.

Platform Notes **Windows:** On Win16 platforms, *localname$* is ignored.

New (keyword)

Syntax 1 `Dim` *ObjectVariable* `As New` *ObjectType*

Syntax 2 `Set` *ObjectVariable* `= New` *ObjectType*

Description Creates a new instance of the specified object type, assigning it to the specified object variable.

Comments The **New** keyword is used to declare a new instance of the specified data object. This keyword can only be used with data object types.

At runtime, the application or extension that defines that object type is notified that a new object is being defined. The application responds by creating a new physical object (within the appropriate context) and returning a reference to that object, which is immediately assigned to the variable being declared.

When that variable goes out of scope (i.e., the **Sub** or **Function** procedure in which the variable is declared ends), the application is notified. The application then performs some appropriate action, such as destroying the physical object.

See Also **Dim** (statement); **Set** (statement).

Platform(s) All.

Not (operator)

Syntax `Not` *expression*

Description Returns either a logical or binary negation of *expression.*

Comments The result is determined as shown in the following table:

If the expression is	then the result is
True	**False**
False	**True**
Null	**Null**
Any numeric type	A binary negation of the number. If the number is an **Integer**, then an **Integer** is returned. Otherwise, the expression is first converted to a **Long**, then a binary negation is performed, returning a **Long**.
Empty	Treated as a **Long** value 0.

Example

```
'This example demonstrates the use of the Not operator in
'comparing logical expressions and for switching a True/False
'toggle variable.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a = False
   b = True
   If (Not a and b) Then message = "a = False, b = True" & crlf

   toggle% = True
   message = message & "toggle% is now " _
      & Format(toggle%,"True/False") & crlf
   toggle% = Not toggle%
   message = message & "toggle% is now " _
      & Format(toggle%,"True/False") & crlf
```

```
   toggle% = Not toggle%
   message = message & "toggle% is now " _
      & Format(toggle%,"True/False")
   MsgBox message
End Sub
```

See Also **Boolean** (data type); Comparison Operators (topic).

Platform(s) All.

Now (function)

Syntax `Now[()]`

Description Returns a **Date** variant representing the current date and time.

Example

```
'This example shows how the Now function can be used as an
'elapsed-time counter.
Sub Main()
   t1# = Now()
   MsgBox "Wait a while and click OK."
   t2# = Now()
   t3# = Second(t2#) - Second(t1#)
   MsgBox "Elapsed time was: " & t3# & " seconds."
End Sub
```

See Also **Date**, **Date$** (functions); **Time**, **Time$** (functions).

Platform(s) All.

NPer (function)

Syntax `NPer(`*rate*`, `*pmt*`, `*pv*`, `*fv*`, `*due*`)`

Description Returns the number of periods for an annuity based on periodic fixed payments and a constant rate of interest.

Comments An annuity is a series of fixed payments paid to or received from an investment over a period of time. Examples of annuities are mortgages, retirement plans, monthly savings plans, and term loans.

The **NPer** function requires the following named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period. If the periods are monthly, be sure to normalize annual rates by dividing them by 12.

Named Parameter	Description
pmt	**Double** representing the amount of each payment or income. Income is represented by positive values, whereas payments are represented by negative values.
pv	**Double** representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan, and the future value (see below) would be zero.
fv	**Double** representing the future value of your annuity. In the case of a loan, the future value would be zero, and the present value would be the amount of the loan.
due	**Integer** indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 indicates payment at the start of each period.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example

```
'This example calculates the number of $100.00 monthly payments
'necessary to accumulate $10,000.00 at an annual rate of 10%.
'Payments are made at the beginning of the month.
Sub Main()
   ag# = NPer((.10/12),100,0,10000,1)
   MsgBox "The number of monthly periods is: " &
Format(ag#,"Standard")
End Sub
```

See Also **IPmt** (function); **Pmt** (function); **PPmt** (function); **Rate** (function).

Platform(s) All.

Npv (function)

Syntax `Npv(`*rate*`, `*valuearray()*`)`

Description Returns the net present value of an annuity based on periodic payments and receipts, and a discount rate.

Comments The **Npv** function requires the following named parameters:

Named Parameter	Description
rate	**Double** that represents the interest rate over the length of the period. If the values are monthly, annual rates must be divided by 12 to normalize them to monthly rates.

Named Parameter	Description
valuearray()	Array of **Double** numbers representing the payments and receipts. Positive values are payments, and negative values are receipts. There must be at least one positive and one negative value.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

For accurate results, be sure to enter your payments and receipts in the correct order because **Npv** uses the order of the array values to interpret the order of the payments and receipts.

If your first cash flow occurs at the beginning of the first period, that value must be added to the return value of the **Npv** function. It should not be included in the array of cash flows.

Npv differs from the **Pv** function in that the payments are due at the end of the period and the cash flows are variable. **Pv**'s cash flows are constant, and payment may be made at either the beginning or end of the period.

Example

```
'This example illustrates the purchase of a lemonade stand for
'$800 financed with money borrowed at 10%. The returns are
'estimated to accelerate as the stand gains popularity. The
'incomes are estimated (generated) over 12 months. This program
'first generates the income stream array in two For...Next loops,
'and then the net present value (Npv) is calculated and displayed.
'Note normalization of the annual 10% rate.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim valu#(12)
   valu(1) = -800                'Initial investment
   message = valu(1) & ", "
   For x = 2 To 5                'Months 2-5
      valu(x) = 100 + (x * 2)
      message = message & valu(x) & ", "
   Next x
   For x = 6 To 12               'Months 6-12
      valu(x) = 100 + (x * 10)'Accelerated income
      message = message & valu(x) & ", "
   Next x
   NetVal# = NPV((.10/12),valu)
   message = "The values:" & crlf & message & crlf & crlf
   MsgBox message & "Net present value: " _
      & Format(NetVal#,"Currency")
End Sub
```

See Also **Fv** (function); **IRR** (function); **MIRR** (function); **Pv** (function).

Platform(s) All.

Object (data type)

Syntax `Object`

Description A data type used to declare OLE Automation variables.

Comments The **Object** type is used to declare variables that reference objects within an application using OLE Automation.

Each object is a 4-byte (32-bit) value that references the object internally. The value 0 (or **Nothing**) indicates that the variable does not reference a valid object, as is the case when the object has not yet been given a value. Accessing properties or methods of such **Object** variables generates a runtime error.

Using Objects

Object variables are declared using the **Dim**, PUBLIC, or **Private** statement:

```
Dim MyApp As Object
```

Object variables can be assigned values (thereby referencing a real physical object) using the **Set** statement:

```
Set MyApp = CreateObject("phantom.application")
Set MyApp = Nothing
```

Properties of an **Object** are accessed using the dot (.) separator:

```
MyApp.Color = 10
i% = MyApp.Color
```

Methods of an **Object** are also accessed using the dot (.) separator:

```
MyApp.Open "sample.txt"
isSuccess = MyApp.Save("new.txt",15)
```

Automatic Destruction

BasicScript keeps track of the number of variables that reference a given object so that the object can be destroyed when there are no longer any references to it:

```
Sub Main()              'Number of references to object
   Dim a As Object                              '0
   Dim b As Object                              '0
   Set a = CreateObject("phantom.application)   '1
   Set b = a                                    '2
   Set a = Nothing                              '1
End Sub                                         '0 (object
```

```
                                                    'destroyed)
```

Note: An OLE Automation object is instructed by BasicScript to destroy itself when no variables reference that object. However, it is the responsibility of the OLE Automation server to destroy it. Some servers do not destroy their objects, usually when the objects have a visual component and can be destroyed manually by the user.

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Single** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement).

Platform(s) Windows, Win32, Macintosh.

Objects (topic)

BasicScript defines two types of objects: data objects and OLE Automation objects.

Syntactically, these are referenced in the same way.

What Is an Object

An object in BasicScript is an encapsulation of data and routines into a single unit. The use of objects in BasicScript has the effect of grouping together a set of functions and data items that apply only to a specific object type.

Objects expose data items for programmability called properties. For example, a sheet object may expose an integer called **NumColumns**. Usually, properties can be both retrieved (get) and modified (set).

Objects also expose internal routines for programmability called methods. In BasicScript, an object method can take the form of a function or a subroutine. For example, a OLE Automation object called **MyApp** may contain a method subroutine called **Open** that takes a single argument (a filename), as shown below:

```
MyApp.Open "c:\files\sample.txt"
```

Declaring Object Variables

In order to gain access to an object, you must first declare an object variable using either **Dim**, **Public**, or **Private**:

```
Dim o As Object   'OLE Automation object
```

Initially, objects are given the value 0 (or **Nothing**). Before an object can be accessed, it must be associated with a physical object.

Assigning a Value to an Object Variable

An object variable must reference a real physical object before accessing any properties or methods of that object. To instantiate an object, use the **Set** statement.

```
Dim MyApp As Object
Set MyApp = CreateObject("Server.Application")
```

Accessing Object Properties

Once an object variable has been declared and associated with a physical object, it can be modified using BasicScript code. Properties are syntactically accessible using the dot operator, which separates an object name from the property being accessed:

```
MyApp.BackgroundColor = 10
i% = MyApp.DocumentCount
```

Properties are set using BasicScript's normal assignment statement:

```
MyApp.BackgroundColor = 10
```

Object properties can be retrieved and used within expressions:

```
i% = MyApp.DocumentCount + 10
MsgBox "Number of documents = " & MyApp.DocumentCount
```

Accessing Object Methods

Like properties, methods are accessed via the dot operator. Object methods that do not return values behave like subroutines in BasicScript (i.e., the arguments are not enclosed within parentheses):

```
MyApp.Open "c:\files\sample.txt",True,15
```

Object methods that return a value behave like function calls in BasicScript. Any arguments must be enclosed in parentheses:

```
If MyApp.DocumentCount = 0 Then MsgBox "No open documents."
NumDocs = app.count(4,5)
```

There is no syntactic difference between calling a method function and retrieving a property value, as shown below:

```
variable = object.property(arg1,arg2)
variable = object.method(arg1,arg2)
```

Comparing Object Variables

The values used to represent objects are meaningless to the script in which they are used, with the following exceptions:

- Objects can be compared to each other to determine whether they refer to the same object.
- Objects can be compared with **Nothing** to determine whether the object variable refers to a valid object.

Object comparisons are accomplished using the **Is** operator:

```
If a Is b Then MsgBox "a and b are the same object."
If a Is Nothing Then MsgBox "a is not initialized."
If b Is Not Nothing Then MsgBox "b is in use."
```

Collections

A collection is a set of related object variables. Each element in the set is called a member and is accessed via an index, either numeric or text, as shown below:

```
MyApp.Toolbar.Buttons(0)
MyApp.Toolbar.Buttons("Tuesday")
```

It is typical for collection indexes to begin with 0.

Each element of a collection is itself an object, as shown in the following examples:

```
Dim MyToolbarButton As Object
Set MyToolbarButton = MyApp.Toolbar.Buttons("Save")
MyAppp.Toolbar.Buttons(1).Caption = "Open"
```

The collection itself contains properties that provide you with information about the collection and methods that allow navigation within that collection:

```
Dim MyToolbarButton As Object
NumButtons% = MyApp.Toolbar.Buttons.Count
MyApp.Toolbar.Buttons.MoveNext
MyApp.Toolbar.Buttons.FindNext "Save"
For i = 1 To MyApp.Toolbar.Buttons.Count
   Set MyToolbarButton = MyApp.Toolbar.Buttons(i)
   MyToolbarButton.Caption = "Copy"
Next i
```

Predefined Objects

BasicScript predefines a few objects for use in all scripts. These are:

```
Clipboard       System          Desktop         HWND
Net             Basic           Screen
```

Note: Some of these objects are not available on all platforms.

Oct, Oct$ (functions)

Syntax `Oct[$](`*number*`)`

Description Returns a **String** containing the octal equivalent of the specified number.

Comments **Oct$** returns a **String**, whereas **Oct** returns a **String** variant.

The returned string contains only the number of octal digits necessary to represent the number.

The *number* parameter is any numeric expression. If this parameter is **Null**, then **Null** is returned. **Empty** is treated as 0. The *number* parameter is rounded to the nearest whole number before converting to the octal equivalent.

Example

```
'This example displays the octal equivalent of several numbers.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   st$ = "The octal values are: " & crlf
   For x = 1 To 5
      y% = x * 10
      st$ = st$ & y% & " : " & Oct$(y%) & crlf
   Next x
   MsgBox st$
End Sub
```

See Also **Hex**, **Hex$** (functions).

Platform(s) All.

OKButton (statement)

Syntax `OKButton` *x*,*y*,*width*,*height* [,*.Identifier*]

Description Creates an OK button within a dialog box template.

Comments This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **OKButton** statement accepts the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

If the *DefaultButton* parameter is not specified in the **Dialog** statement, the OK button will be used as the default button. In this case, the OK button can be selected by pressing Enter on a nonbutton control.

A dialog box template must contain at least one **OKButton**, **CancelButton**, or **PushButton** statement (otherwise, the dialog box cannot be dismissed).

Example

```
'This example shows how to use the OK and Cancel buttons within a
'dialog box template and how to detect which one closed the dialog
'box.
Sub Main()
   Begin Dialog ButtonTemplate 17,33,104,23,"Buttons"
      OKButton 8,4,40,14,.OK
      CancelButton 56,4,40,14,.Cancel
   End Dialog
   Dim ButtonDialog As ButtonTemplate
   WhichButton = Dialog(ButtonDialog)
   If WhichButton = -1 Then
      MsgBox "OK was pressed."
   ElseIf WhichButton = 0 Then
      MsgBox "Cancel was pressed."
   End If
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

On Error (statement)

Syntax `On Error {Goto` *label* `| Resume Next | Goto 0}`

Description Defines the action taken when a trappable runtime error occurs.

Comments The form **On Error Goto** *label* causes execution to transfer to the specified label when a runtime error occurs.

The form **On Error Resume Next** causes execution to continue on the line following the line that caused the error.

The form **On Error Goto 0** causes any existing error trap to be removed.

If an error trap is in effect when the script ends, then an error will be generated.

An error trap is only active within the subroutine or function in which it appears.

Once an error trap has gained control, appropriate action should be taken, and then control should be resumed using the **Resume** statement. The **Resume** statement resets the error handler and continues execution. If a procedure ends while an error is pending, then an error will be generated. (The **Exit Sub** or **Exit Function** statement also resets the error handler, allowing a procedure to end without displaying an error message.)

Errors within an Error Handler

If an error occurs within the error handler, then the error handler of the caller (or any procedure in the call stack) will be invoked. If there is no such error handler, then the error is fatal, causing the script to stop executing. The following statements reset the error state (i.e., these statements turn off the fact that an error occurred):

```
Resume
Err=-1
```

The **Resume** statement forces execution to continue either on the same line or on the line following the line that generated the error. The **Err=-1** statement allows explicit resetting of the error state so that the script can continue normal execution without resuming at the statement that caused the error condition.

The **On Error** statement will not reset the error. Thus, if an **On Error** statement occurs within an error handler, it has the effect of changing the location of a new error handler for any new errors that may occur once the error has been reset.

Example

```
'This example will demonstrate three types of error handling. The
'first case simply by-passes an expected error and continues with
'program operation. The second case creates an error branch that
'jumps to a common error handling routine that processes incoming
'errors, clears the error (with the Resume statement) and resumes
'program execution. The third case clears all internal error
'handling so that execution will stop when the next error is
'encountered.
Sub Main()
   Dim x%
   a = 10000
   b = 10000

   On Error Goto Pass'Branch to this label on error.
   Do
      x% = a * b
   Loop

Pass:
   Err = -1          'Clear error status.
   MsgBox "Cleared error status and continued."

   On Error Goto Overflow  'Branch to new error routine on any
   x% = 1000               'subsequent errors.
```

```
   x% = a * b
   x% = a / 0

   On Error Goto 0   'Clear error branching.
   x% = a * b        'Program will stop here.
   Exit Sub          'Exit before common error routine.

Overflow:            'Beginning of common error routine.
   If Err = 6 then
      MsgBox "Overflow Branch."
   Else
      MsgBox Error(Err)
   End If
   Resume Next
End Sub
```

See Also Error Handling (topic); **Error** (statement); **Resume** (statement).

Platform(s) All.

Open (statement)

Syntax `Open` *filename$* `[For` *mode*`] [Access` *accessmode*`] [`*lock*`] As [#]` *filenumber* _
`[Len =` *reclen*`]`

Description Opens a file for a given mode, assigning the open file to the supplied *filenumber*.

Comments The *filename$* parameter is a string expression that contains a valid filename.

The *filenumber* parameter is a number between 1 and 255. The **FreeFile** function can be used to determine an available file number.

The *mode* parameter determines the type of operations that can be performed on that file:

File Mode	Description
Input	Opens an existing file for sequential input (*filename$* must exist). The value of *accessmode*, if specified, must be **Read**.
Output	Opens an existing file for sequential output, truncating its length to zero, or creates a new file. The value of *accessmode*, if specified, must be **Write**.
Append	Opens an existing file for sequential output, positioning the file pointer at the end of the file, or creates a new file. The value of *accessmode*, if specified, must be **Read Write**.

File Mode	Description
Binary	Opens an existing file for binary I/O or creates a new file. Existing binary files are never truncated in length. The value of *accessmode*, if specified, determines how the file can subsequently be accessed.
Random	Opens an existing file for record I/O or creates a new file. Existing random files are truncated only if *accessmode* is **Write**. The *reclen* parameter determines the record length for I/O operations.

If the *mode* parameter is missing, then **Random** is used.

The *accessmode* parameter determines what type of I/O operations can be performed on the file:

Access	Description
Read	Opens the file for reading only. This value is valid only for files opened in **Binary**, **Random**, or **Input** mode.
Write	Opens the file for writing only. This value is valid only for files opened in **Binary**, **Random**, or **Output** mode.
Read Write	Opens the file for both reading and writing. This value is valid only for files opened in **Binary**, **Random**, or **Append** mode.

If the *accessmode* parameter is not specified, the following defaults are used:

File Mode	Default Value for *accessmode*
Input	**Read**
Output	**Write**
Append	**Read Write**
Binary	When the file is initially opened, access is attempted three times in the following order: **1. Read Write** **2. Write** **3. Read**
Random	Same as **Binary** files

The *lock* parameter determines what access rights are granted to other processes that attempt to open the same file. The following table describes the values for *lock*:

lock Value	Description
Shared	Another process can both read this file and write to it. (Deny none.)

lock Value	Description
Lock Read	Another process can write to this file but not read it. (Deny read.)
Lock Write	Another process can read this file but not write to it. (Deny write.)
Lock Read Write	Another process is prevented both from reading this file and from writing to it. (Exclusive.)

If *lock* is not specified, then the file is opened in **Shared** mode.

If the file does not exist and the *lock* parameter is specified, the file is opened twice—once to create the file and again to establish the correct sharing mode.

Files opened in **Random** mode are divided up into a sequence of records, each of the length specified by the *reclen* parameter. If this parameter is missing, then 128 is used. For files opened for sequential I/O, the *reclen* parameter specifies the size of the internal buffer used by BasicScript when performing I/O. Larger buffers mean faster file access. For **Binary** files, the *reclen* parameter is ignored.

For files opened in **Append** mode, BasicScript opens the file and positions the file pointer after the last character in the file. The end-of-file character, if present, is not removed by BasicScript.

Example

```
'This example opens several files in various configurations.
Sub Main()
   Open "test.dat" For Output Access Write Lock Write As #2
   Close
   Open "test.dat" For Input Access Read Shared As #1
   Close
   Open "test.dat" For Append Access Write Lock Read Write as #3
   Close
   Open "test.dat" For Binary Access Read Write Shared As #4
   Close
   Open "test.dat" For Random Access Read Write Lock Read As #5
   Close
   Open "test.dat" For Input Access Read Shared As #6
   Close
   Kill "test.dat"
End Sub
```

See Also **Close** (statement); **Reset** (statement); **FreeFile** (function).

Platform(s) All.

Platform Notes **UNIX:** BasicScript sets the permissions of new files to the logical conjunction of 0777 octal and the process's umask.

OpenFileName$ (function)

Syntax `OpenFileName$[([title$ [,[extensions$] [,helpfile,context]]])]`

Description Displays a dialog box that prompts the user to select from a list of files, returning the full pathname of the file the user selects or a zero-length string if the user selects Cancel.

Comments This function displays the standard file open dialog box, which allows the user to select a file. It takes the following parameters:

Parameter	Description
title$	**String** specifying the title that appears in the dialog box's title bar. If this parameter is omitted, then "Open" is used.
extension$	**String** specifying the available file types. The format for this string depends on the platform on which BasicScript is running. If this parameter is omitted, then all files are displayed.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Example

```
'This example asks the user for the name of a file, then proceeds
'to read the first line from that file.

Sub Main
   Dim f As String,s As String
   f$ = OpenFileName$("Open Picture","Text Files:*.TXT")
   If f$ <> "" Then
      Open f$ For Input As #1
      Line Input #1,s$
      Close #1
      MsgBox "First line from " & f$ & " is " & s$
   End If
End Sub
```

See Also **MsgBox** (statement); **AskBox, AskBox$** (functions); **AskPassword**, **AskPassword$** (functions); **InputBox**, **InputBox$** (functions); **SaveFileName$** (function); **SelectBox** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32, OS/2:** The *extensions$* parameter must be in the following format:

type:*ext*[,*ext*][;*type*:*ext*[,*ext*]]...

Placeholder	Description
type	Specifies the name of the grouping of files, such as All Files.
ext	Specifies a valid file extension, such as ***.BAT** or ***.?F?**.

For example, the following are valid *extensions$* specifications:

```
"All Files:*.*"
"Documents:*.TXT,*.DOC"
"All Files:*.*;Documents:*.TXT,*.DOC"
```

Macintosh: On the Macintosh, the *extensions$* parameter contains a comma-separated list of four-character file types. For example:

```
"TEXT,XLS4,MSWD"
```

On the Macintosh, the *title$* parameter is ignored.

Operator Precedence (topic)

The following table shows the precedence of the operators supported by BasicScript. Operations involving operators of higher precedence occur before operations involving operators of lower precedence. When operators of equal precedence occur together, they are evaluated from left to right.

Operator	Description	Precedence Order
()	Parentheses	Highest
^	Exponentiation	
-	Unary minus	
/, *	Division and multiplication	
\	Integer division	
Mod	Modulo	
+, -	Addition and subtraction	
&	String concatenation	
=, <>, >, <, <=, >=	Relational	
Like, **Is**	String and object comparison	
Not	Logical negation	
And	Logical or binary conjunction	

Operator	Description	Precedence Order
Or	Logical or binary disjunction	
Xor, **Eqv**, **Imp**	Logical or binary operators	Lowest

The precedence order can be controlled using parentheses, as shown below:

```
a = 4 + 3 * 2        'a becomes 10.
a = (4 + 3) * 2      'a becomes 14.
```

Operator Precision (topic)

When numeric, binary, logical or comparison operators are used, the data type of the result is generally the same as the data type of the more precise operand. For example, adding an **Integer** and a **Long** first converts the **Integer** operand to a **Long**, then preforms a long addition, overflowing only if the result cannot be contained with a **Long**. The order of precision is shown in the following list:

Empty	Least precise
Boolean	
Integer	
Long	
Single	
Date	
Double	
Currency	Most precise

There are exceptions noted in the descriptions of each operator.

The rules for operand conversion are further complicated when an operator is used with variant data. In many cases, an overflow causes automatic promotion of the result to the next highest precise data type. For example, adding two **Integer** variants results in an **Integer** variant unless it overflows, in which case the result is automatically promoted to a **Long** variant.

Option Base (statement)

Syntax `Option Base {0 | 1}`

Description Sets the lower bound for array declarations.

Comments By default, the lower bound used for all array declarations is 0.

This statement must appear outside of any functions or subroutines.

Example

```
Option Base 1
Sub Main()
   Dim a(10)      'Contains 10 elements (not 11).
End Sub
```

See Also **Dim** (statement); **Public** (statement); **Private** (statement).

Platform(s) All.

Option Compare (statement)

Syntax `Option Compare [Binary | Text]`

Description Controls how strings are compared.

Comments When **Option Compare** is set to **Binary**, then string comparisons are case-sensitive (e.g., "A" does not equal "a"). When it is set to **Text**, string comparisons are case-insensitive (e.g., "A" is equal to "a").

The default value for **Option Compare** is **Binary**.

The **Option Compare** statement affects all string comparisons in any statements that follow the **Option Compare** statement. Additionally, the setting affects the default behavior of **Instr**, **StrComp**, and the **Like** operator. The following table shows the types of string comparisons affected by this setting:

>	<	<>
<=	>=	**Instr**
StrComp	**Like**	

The **Option Compare** statement must appear outside the scope of all subroutines and functions. In other words, it cannot appear within a **Sub** or **Function** block.

Example

```
'This example shows the use of Option Compare.
Option Compare Binary
Sub CompareBinary
   a$ = "This String Contains UPPERCASE."
   b$ = "this string contains uppercase."
   If a$ = b$ Then
      MsgBox "The two strings were compared case-insensitive."
   Else
      MsgBox "The two strings were compared case-sensitive."
```

```
   End If
End Sub
Option Compare Text
Sub CompareText
   a$ = "This String Contains UPPERCASE."
   b$ = "this string contains uppercase."
   If a$ = b$ Then
      MsgBox "The two strings were compared case-insensitive."
   Else
      MsgBox "The two strings were compared case-sensitive."
   End If
End Sub
Sub Main()
   CompareBinary  'Calls subroutine above.
   CompareText    'Calls subroutine above.
End Sub
```

See Also **Like** (operator); **InStr**, **InStrB** (functions); **StrComp** (function); Comparison Operators (topic).

Platform(s) All.

Option CStrings (statement)

Syntax `Option CStrings {On | Off}`

Description Turns on or off the ability to use C-style escape sequences within strings.

Comments When **Option CStrings On** is in effect, the compiler treats the backslash character as an escape character when it appears within strings. An escape character is simply a special character that otherwise cannot ordinarily be typed by the computer keyboard.

Escape	Description	Equivalent Expression
\r	Carriage return	**Chr$(13)**
\n	Line Feed	**Chr$(10)**
\a	Bell	**Chr$(7)**
\b	Backspace	**Chr$(8)**
\f	Form Feed	**Chr$(12)**
\t	Tab	**Chr$(9)**
\v	Vertical tab	**Chr$(11)**
\0	Null	**Chr$(0_**
\"	Double quote	"" or **Chr$(34)**
\\	Backslash	**Chr$(92)**

Escape	Description	Equivalent Expression
?	Question mark	**?**
'	Single quote	**'**
x*hh*	Hexadecimal number	**Chr$(Val(&H***hh***))**
ooo	Octal number	**Chr$(Val(&O***ooo***))**
anycharacter	Any character	*anycharacter*

With hexadecimal values, BasicScript stops scanning for digits when it encounters a nonhexadecimal digit or two digits, whichever comes first. Similarly, with octal values, BasicScript stops scanning when it encounters a nonoctal digit or three digits, whichever comes first.

When **Option CStrings Off** is in effect, then the backslash character has no special meaning. This is the default.

Example

```
Option CStrings On
Sub Main()
   MsgBox "They said, \"Watch out for that clump of grass!\""
   MsgBox "First line.\r\nSecond line."
   MsgBox "Char A: \x41 \r\n Char B: \x42"
End Sub
```

Platform(s) All.

Option Default (statement)

Syntax `Option Default` *type*

Description Sets the default data type of variables and function return values when not otherwise specified.

Comments By default, the type of implicitly defined variables and function return values is **Variant**. This statement is used for backward compatibility with earlier versions of BasicScript where the default data type was **Integer**.

This statement must appear outside the scope of all functions and subroutines.

Currently, *type* can only be set to **Integer**.

Example

```
'This script sets the default data type to Integer. This fact
'is used to declare the function AddIntegers which returns an
'Integer data type.
Option Default Integer
Function AddIntegers(a As Integer,b As Integer)
   Foo = a + b
End Function
```

```
Sub Main
   Dim a,b,result
   a = InputBox("Enter an integer:")
   b = InputBox("Enter an integer:")
   result = AddIntegers(a,b)
End Sub
```

See Also **DefType** (statement).

Platform(s) All.

Option Explicit (statement)

Syntax `Option Explicit`

Description Prevents implicit declaration of variables and externally called procedures.

Comments By default, BasicScript implicitly declares variables that are used but have not been explicitly declared with **Dim**, **Public**, or **Private**. To avoid typing errors, you may want to use **Option Explicit** to prevent this behavior.

The **Option Explicit** statement also enforces explicit declaration of all externally called procedures. Once specified, all externally called procedures must be explicitly declared with the **Declare** statement.

See Also **Const** (statement); **Dim** (statement); **Public** (statement); **Private** (statement); **ReDim** (statement); **Declare** (statement).

Platform(s) All.

OptionButton (statement)

Syntax `OptionButton` *x*`,`*y*`,`*width*`,`*height*`,`*title$* `[,`*.Identifier*`]`

Description Defines an option button within a dialog box template.

Comments This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **OptionButton** statement accepts the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** coordinates specifying the dimensions of the control in dialog units.

Parameter	Description
title$	**String** containing text that appears within the option button. This text may contain an ampersand character to denote an accelerator letter, such as "&Portrait" for Portrait, which can be selected by pressing the P accelerator.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

Example See **OptionGroup** (statement).

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32, OS/2:** On Windows, Win32, and OS/2 platforms, accelerators are underlined, and the accelerator combination Alt+*letter* is used.

Macintosh: On the Macintosh, accelerators are normal in appearance, and the accelerator combination Command+*letter* is used.

OptionEnabled (function)

Syntax `OptionEnabled(`*name$ | id*`)`

Description Returns **True** if the specified option button is enabled within the current window or dialog box; returns **False** otherwise.

Comments This function is used to determine whether a given option button is enabled within the current window or dialog box. If an option button is enabled, then its value can be set using the **SetOption** statement.

The **OptionEnabled** statement takes the following parameters:

Parameter	Description
name$	**String** containing the name of the option button.
id	**Integer** specifying the ID of the option button.

Note: The **OptionEnabled** function is used to determine whether an option button is enabled in another application's dialog box. Use the **DlgEnable** function with dynamic dialog boxes.

Example

```
'This example checks to see whether the option button is enabled
'before setting it.
If OptionEnabled("Tile") Then
   SetOption "Tile"
End If
```

See Also **GetOption** (function); **OptionExists** (function); **SetOption** (statement).

Platform(s) Windows.

OptionExists (function)

Syntax `OptionExists(`*name$* `|` *id*`)`

Description Returns **True** if the specified option button exists within the current window or dialog box; returns **False** otherwise.

Comments This function is used to determine whether a given option button exists within the current window or dialog box.

The **OptionExists** statement takes the following parameters:

Parameter	Description
name$	**String** containing the name of the option button.
id	**Integer** specifying the ID of the option button.

Note: The **OptionExists** function is used to determine whether an option button exists in another application's dialog box. There is no equivalent function for use with dynamic dialog boxes.

Example

```
'This example checks to see whether the option button exists and
'is enabled before setting it.
If OptionExists("Tile") Then
   If OptionEnabled("Tile") Then
      SetOption("Tile")
   End If
End If
```

See Also **GetOption** (function); **OptionEnabled** (function); **SetOption** (statement).

Platform(s) Windows.

OptionGroup (statement)

Syntax `OptionGroup .`*Identifier*

Description Specifies the start of a group of option buttons within a dialog box template.

Comments The *.Identifier* parameter specifies the name by which the group of option buttons can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates an integer variable whose value corresponds to the index of the selected option button within the group (0 is the first option button, 1 is the second option button, and so on). This variable can be accessed using the following syntax: *DialogVariable.Identifier*.

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

When the dialog box is created, the option button specified by *.Identifier* will be on; all other option buttons in the group will be off. When the dialog box is dismissed, the *.Identifier* will contain the selected option button.

Example

```
'This example creates a group of option buttons.
Sub Main()
   Begin Dialog PrintTemplate 16,31,128,65,"Print"
      GroupBox 8,8,64,52,"Orientation",.Junk
      OptionGroup .Orientation
         OptionButton 16,20,37,8,"Portrait",.Portrait
         OptionButton 16,32,51,8,"Landscape",.Landscape
         OptionButton 16,44,49,8,"Don't Care",.DontCare
      OKButton 80,8,40,14
   End Dialog
   Dim PrintDialog As PrintTemplate
   Dialog PrintDialog
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Or (operator)

Syntax *result* = *expression1* `Or` *expression2*

Description Performs a logical or binary disjunction on two expressions.

Comments

If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical disjunction is performed as follows:

If *expression1* **is**	**and** *expression2* **is**	**then the** *result* **is**
True	**True**	**True**
True	**False**	**True**
True	**Null**	**True**
False	**True**	**True**
False	**False**	**False**
False	**Null**	**Null**
Null	**True**	**True**
Null	**False**	**Null**
Null	**Null**	**Null**

Binary Disjunction

If the two expressions are **Integer**, then a binary disjunction is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long** and a binary disjunction is then performed, returning a **Long** result.

Binary disjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in *expression1* **is**	**and bit in** *expression2* **is**	**the** *result* **is**
1	1	1
0	1	1
1	0	1
0	0	0

Examples

```
'This first example shows the use of logical Or.
Dim s$ As String
s$ = InputBox$("Enter a string.")
If s$ = "" Or Mid$(s$,1,1) = "A" Then
   s$ = LCase$(s$)
End If
'This second example shows the use of binary Or.
Dim w As Integer
TryAgain:
   s$ = InputBox$("Enter a hex number (four digits max).")
   If Mid$(s$,1,1) <> "&" Then
      s$ = "&H" & s$
   End If
```

```
If Not IsNumeric(s$) Then Goto TryAgain
w = CInt(s$)
MsgBox "Your number is &H" & Hex$(w)
w = w Or &H8000
MsgBox "Your number with the high bit set is &H" & Hex$(w)
```

See Also Operator Precedence (topic); **Xor** (operator); **Eqv** (operator); **Imp** (operator); **And** (operator).

Platform(s) All.

Picture (statement)

Syntax `Picture` *x*`,`*y*`,`*width*`,`*height*`,`*PictureName$*`,`**`PictureType`** `[,[`.*Identifier*`] [,`*style*`]]`

Description Creates a picture control in a dialog box template.

Comments Picture controls are used for the display of graphics images only. The user cannot interact with these controls.

The **Picture** statement accepts the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the dimensions of the control in dialog units.
PictureName$	**String** containing the name of the picture. If *PictureType* is 0, then this name specifies the name of the file containing the image. If *PictureType* is 10, then *PictureName$* specifies the name of the image within the resource of the picture library. If *PictureName$* is empty, then no picture will be associated with the control. A picture can later be placed into the picture control using the **DlgSetPicture** statement.
PictureType	**Integer** specifying the source for the image. The following sources are supported: 0 - The image is contained in a file on disk. 10 - The image is contained in a picture library as specified by the *PicName$* parameter on the **Begin Dialog** statement.
.*Identifier*	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). If omitted, then the first two words of *PictureName$* are used.
style	Specifies whether the picture is drawn within a 3D frame. It can be either of the following values: 0 - Draw the picture control with a normal frame. 1 - Draw the picture control with a 3D frame. If this parameter is omitted, then the picture control is drawn with a normal frame.

The picture control extracts the actual image from either a disk file or a picture library. In the case of bitmaps, both 2- and 16-color bitmaps are supported. In the case of WMFs, BasicScript supports the Placeable Windows Metafile.

If *PictureName$* is a zero-length string, then the picture is removed from the picture control, freeing any memory associated with that picture.

Examples

```
'This first example shows how to use a picture from a file.
Sub Main()
   Begin Dialog LogoDialogTemplate 16,32,288,76,"Introduction"
      OKButton 240,8,40,14
      Picture 8,8,224,64,"c:\bitmaps\logo.bmp",0,.Logo
   End Dialog
   Dim LogoDialog As LogoDialogTemplate
   Dialog LogoDialog
End Sub
'This second example shows how to use a picture from a picture
'library with a 3D frame.
Sub Main()
   Begin Dialog LogoDlg 16,31,288,76,"Introduction",,"pics.dll"
      OKButton 240,8,40,14
      Picture 8,8,224,64,"CompanyLogo",10,.Logo,1
   End Dialog
   Dim LogoDialog As LogoDialogTemplate
   Dialog LogoDialog
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **DlgSetPicture** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32:** Picture controls can contain either a bitmap or a WMF (Windows metafile). When extracting images from a picture library, BasicScript assumes that the resource type for metafiles is 256.

Picture libraries are implemented as DLLs on the Windows and Win32 platforms.

OS/2: Picture controls can contain either bitmaps or Windows metafiles.

Picture libraries under OS/2 are implemented as resources within DLLs. The *PictureName$* parameter corresponds to the name of one of these resources as it appears within the DLL.

Macintosh: Picture controls on the Macintosh can contain only PICT images. These are contained in files of type PICT.

Picture libraries on the Macintosh are files with collections of named PICT resources. The *PictureName$* parameter corresponds to the name of one the resources as it appears within the file.

PictureButton (statement)

Syntax `PictureButton` *x*,*y*,*width*,*height*,*PictureName$*,*PictureType* [,*.Identifier*]

Description Creates a picture button control in a dialog box template.

Comments Picture button controls behave very much like push button controls. Visually, picture buttons are different from push buttons in that they contain a graphic image imported either from a file or from a picture library.

The **PictureButton** statement accepts the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** coordinates specifying the dimensions of the control in dialog units.
PictureName$	**String** containing the name of the picture. If *PictureType* is 0, then this name specifies the name of the file containing the image. If *PictureType* is 10, then *PictureName$* specifies the name of the image within the resource of the picture library. If *PictureName$* is empty, then no picture will be associated with the control. A picture can later be placed into the picture control using the **DlgSetPicture** statement.
PictureType	**Integer** specifying the source for the image. The following sources are supported: 0 - The image is contained in a file on disk. 10 - The image is contained in a picture library as specified by the *PicName$* parameter on the **Begin Dialog** statement.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

The picture button control extracts the actual image from either a disk file or a picture library, depending on the value of *PictureType*. The supported picture formats vary from platform to platform.

If *PictureName$* is a zero-length string, then the picture is removed from the picture button control, freeing any memory associated with that picture.

Examples

```
'This first example shows how to use a picture from a file.
Sub Main()
   Begin Dialog LogoDialogTemplate 16,32,288,76,"Introduction"
      OKButton 240,8,40,14
      PictureButton 8,4,224,64,"c:\bitmaps\logo.bmp",0,.Logo
   End Dialog
   Dim LogoDialog As LogoDialogTemplate
   Dialog LogoDialog
End Sub

'This second example shows how to use a picture from a picture
'library.
Sub Main()
   Begin Dialog LogoDlg 16,31,288,76,"Introduction",,"pics.dll"
      OKButton 240,8,40,14
      PictureButton 8,4,224,64,"CompanyLogo",10,.Logo
   End Dialog
   Dim LogoDialog As LogoDlg
   Dialog LogoDialog
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **PushButton** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **Picture** (statement); **DlgSetPicture** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, OS/2, Macintosh, UNIX.

Platform Notes **Windows, Win32:** Picture controls can contain either a bitmap or a WMF (Windows metafile). When extracting images from a picture library, BasicScript assumes that the resource type for metafiles is 256.

Picture libraries are implemented as DLLs on the Windows and Win32 platforms.

OS/2: Picture controls can contain either bitmaps or Windows metafiles.

Picture libraries under OS/2 are implemented as resources within DLLs. The *PictureName$* parameter corresponds to the name of one of these resources as it appears within the DLL.

Macintosh: Picture controls on the Macintosh can contain only PICT images. These are contained in files of type PICT.

Picture libraries on the Macintosh are files with collections of named PICT resources. The *PictureName$* parameter corresponds to the name of one the resources as it appears within the file.

Pmt (function)

Syntax `Pmt(`*rate*`, `*nper*`, `*pv*`, `*fv*`, `*due*`)`

Description Returns the payment for an annuity based on periodic fixed payments and a constant rate of interest.

Comments An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans.

The **Pmt** function requires the following named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period. If the periods are given in months, be sure to normalize annual rates by dividing them by 12.
nper	**Double** representing the total number of payments in the annuity.
pv	**Double** representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan.
fv	**Double** representing the future value of your annuity. In the case of a loan, the future value would be 0.
due	**Integer** indicating when payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 specifies payment at the start of each period.

The *rate* and *nper* parameters must be expressed in the same units. If *rate* is expressed in months, then *nper* must also be expressed in months.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example

```
'This example calculates the payment necessary to repay a
'$1,000.00 loan over 36 months at an annual rate of 10%.
'Payments are due at the beginning of the period.
Sub Main()
   x = Pmt((.1/12),36,1000.00,0,1)
   message = "The payment is: "
   MsgBox message & Format(x,"Currency")
End Sub
```

See Also **IPmt** (function); **NPer** (function); **PPmt** (function); **Rate** (function).

Platform(s) All.

PopupMenu (function)

Syntax `PopupMenu(`*`MenuItems$`*`())`

Description Displays a pop-up menu containing the specified items, returning an **Integer** representing the index of the selected item.

Comments If no item is selected (i.e., the pop-up menu is canceled), then a value of 1 less than the lower bound of the array is returned.

This function creates a pop-up menu using the string elements in the given array. Each array element is used as a menu item. A zero-length string results in a separator bar in the menu.

The pop-up menu is created with the upper left corner at the current mouse position.

A runtime error results if *MenuItems$* is not a single-dimension array.

Only one pop-up menu can be displayed at a time. An error will result if another script executes this function while a pop-up menu is visible.

Example

```
Sub Main()
   Dim a$()
   AppList a$
   w% = PopupMenu(a$)
End Sub
```

See Also **SelectBox** (function).

Platform(s) Windows, Win32.

PPmt (function)

Syntax `PPmt(`*`rate, per, nper, pv, fv, due`*`)`

Description Calculates the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

Comments An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans.

The **PPmt** function requires the following named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period.
per	**Double** representing the number of payment periods. The *per* parameter can be no less than 1 and no greater than *nper*.

Named Parameter	Description
nper	**Double** representing the total number of payments in your annuity.
pv	**Double** representing the present value of your annuity. In the case of a loan, the present value would be the amount of the loan.
fv	**Double** representing the future value of your annuity. In the case of a loan, the future value would be 0.
due	**Integer** indicating when payments are due. If this parameter is 0, then payments are due at the end of each period; if it is 1, then payments are due at the start of each period.

The *rate* and *nper* parameters must be in the same units to calculate correctly. If *rate* is expressed in months, then *nper* must also be expressed in months.

Negative values represent payments paid out, whereas positive values represent payments received.

Example

```
'This example calculates the principal paid during each year on
'a loan of $1,000.00 with an annual rate of 10% for a period of
'10 years. The result is displayed as a table containing the
'following information: payment, principal payment, principal
'balance.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   pay = Pmt(.1,10,1000.00,0,1)
   message = "Amortization table for"
   message = message & " 10 years: " & crlf & crlf
   bal = 1000.00
   For per = 1 to 10
      prn = PPmt(.1,per,10,1000,0,0)
      bal = bal + prn
      message = message & Format(pay,"Currency") & "  " _
         & Format$(Prn,"Currency")
      message = message & "  " & Format(bal,"Currency") & crlf
   Next per
   MsgBox message
End Sub
```

See Also **IPmt** (function); **NPer** (function); **Pmt** (function); **Rate** (function).

Platform(s) All.

Print (statement)

Syntax `Print [[{Spc(n) | Tab(n)}][expressionlist][{; | ,}]]`

Description Prints data to an output device.

Comments The actual output device depends on the platform on which BasicScript is running.

The following table describes how data of different types is written:

Data Type	Description
String	Printed in its literal form, with no enclosing quotes.
Any numeric type	Printed with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.
Boolean	Printed as "True" or "False". These keywords are translated as appropriate according to your system's locale.
Date	Printed using the short date format. If either the date or time component is missing, only the provided portion is printed (this is consistent with the "general date" format understood by the **Format**/**Format$** functions).
Empty	Nothing is printed
Null	Prints "Null". This keyword is translated as appropriate according to your system's locale.
User-defined errors	User-defined errors are printed to files as "Error *code*", where *code* is the value of the user-defined error. The word "Error" is not translated. The "Error" keyword is translated as appropriate according to your system's locale.
Object	For any object type, BasicScript retrieves the default property of that object and prints this value using the above rules.

Each expression in *expressionlist* is separated with either a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression in the list is not followed by a comma or a semicolon, then a carriage return is printed to the file. If the last expression ends with a semicolon, no carriage return is printed—the next **Print** statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

The **Tab** and **Spc** functions provide additional control over the column position. The **Tab** function moves the file position to the specified column, whereas the **Spc** function outputs the specified number of spaces.

Note: Null characters—**Chr$(0)**—within strings are translated to spaces when printing to the Viewport window. When printing to files, this translation is not performed.

Examples

```
Sub Main()
   i% = 10
   s$ = "This is a test."
   Print "The value of i=";i%,"the value of s=";s$
   'This example prints the value of i% in print zone 1 and s$
   'in print zone 3.
   Print i%,,s$
   'This example prints the value of i% and s$ separated by 10
   'spaces.
   Print i%;Spc(10);s$
   'This example prints the value of i in column 1 and s$ in
   'column 30.
   Print i%;Tab(30);s$
   'This example prints the value of i% and s$.
   Print i%;s$,
   Print 67
End Sub
```

See Also **Viewport.Open** (method).

Platform(s) All.

Platform Notes **Windows, Win32:** Under Windows, this statement writes data to a viewport window.

If no viewport window is open, then the statement is ignored. Printing information to a viewport window is a convenient way to output debugging information. To open a viewport window, use the following statement:

```
Viewport.Open
```

UNIX, Macintosh: On all UNIX platforms, and the Macintosh, the **Print** statement prints data to **stdout**.

Print# (statement)

Syntax `Print [#]`*filenumber*`, [[{Spc(`*n*`) | Tab(`*n*`)}][`*expressionlist*`][{;|,}]]`

Description Writes data to a sequential disk file.

Comments The *filenumber* parameter is a number that is used by BasicScript to refer to the open file—the number passed to the **Open** statement.

The following table describes how data of different types is written:

Data Type	Description
String	Printed in its literal form, with no enclosing quotes.
Any numeric type	Printed with an initial space reserved for the sign (space = positive). Additionally, there is a space following each number.
Boolean	Printed as "True" or "False". These keywords are translated as appropriate according to your system's locale.
Date	Printed using the short date format. If either the date or time component is missing, only the provided portion is printed (this is consistent with the "general date" format understood by the **Format**/**Format$** functions).
Empty	Nothing is printed
Null	Prints "Null". This keyword is translated as appropriate according to your system's locale.
User-defined errors	User-defined errors are printed to files as "Error *code*", where *code* is the value of the user-defined error. The word "Error" is not translated. The "Error" keyword is translated as appropriate according to your system's locale.
Object	For any object type, BasicScript retrieves the default property of that object and prints this value using the above rules.

Each expression in *expressionlist* is separated with either a comma (,) or a semicolon (;). A comma means that the next expression is output in the next print zone. A semicolon means that the next expression is output immediately after the current expression. Print zones are defined every 14 spaces.

If the last expression in the list is not followed by a comma or a semicolon, then an end-of-line is printed to the file. If the last expression ends with a semicolon, no end-of-line is printed—the next **Print** statement will output information immediately following the expression. If the last expression in the list ends with a comma, the file pointer is positioned at the start of the next print zone on the current line.

The **Write** statement always outputs information ending with an end-of-line. Thus, if a **Print** statement is followed by a **Write** statement, the file pointer is positioned on a new line.

The **Print** statement can only be used with files that are opened in **Output** or **Append** mode.

The **Tab** and **Spc** functions provide additional control over the file position. The **Tab** function moves the file position to the specified column, whereas the **Spc** function outputs the specified number of spaces.

In order to correctly read the data using the **Input#** statement, you should write the data using the **Write** statement.

The end-of-line character is different on many platforms. On some platforms, it is defined as a carriage-return/line-feed pair, and on other platforms, it is defined as only a line feed. The BasicScript statements that read sequential files don't care about the end-of-line character—either will work.

Examples

```
Sub Main()
   'This example opens a file and prints some data.
   Open "test.dat" For Output As #1
   i% = 10
   s$ = "This is a test."
   Print #1,"The value of i=";i%,"the value of s=";s$
   'This example prints the value of i% in print zone 1 and s$
   'in print zone 3.
   Print #1,i%,,s$
   'This example prints the value of i% and s$ separated by ten
   'spaces.
   Print #1,i%;Spc(10);s$
   'This example prints the value of i in column 1 and s$ in
   'column 30.
   Print #1,i%;Tab(30);s$
   'This example prints the value of i% and s$.
   Print #1,i%;s$,
   Print #1,67
   Close #1
   Kill "test.dat"
End Sub
```

See Also **Open** (statement); **Put** (statement); **Write#** (statement).

Platform(s) All.

PrinterGetOrientation (function)

Syntax

```
PrinterGetOrientation[()]
```

Description Returns an **Integer** representing the current orientation of paper in the default printer.

Comments **PrinterGetOrientation** returns **ebPortrait** if the printer orientation is set to portrait; otherwise, it returns **ebLandscape**. Zero is returned if there is no installed default printer.

This function loads the printer driver and therefore may be slow.

Example

```
'This example toggles the printer orientation.
Sub Main()
   If PrinterGetOrientation = ebLandscape Then
```

```
      PrinterSetOrientation ebPortrait
   Else
      PrinterSetOrientation ebLandscape
   End If
End Sub
```

See Also **PrinterSetOrientation** (statement).

Platform(s) Windows.

Windows: The default printer is determined by examining the device= line in the [windows] section of the win.ini file.

PrinterSetOrientation (statement)

Syntax `PrinterSetOrientation` *NewSetting*

Description Sets the orientation of the default printer to *NewSetting*.

Comments The possible values for *NewSetting* are as follows:

Setting	Description
ebLandscape	Sets printer orientation to landscape.
ebPortrait	Sets printer orientation to portrait.

This function loads the printer driver for the default printer and therefore may be slow.

Example See **PrinterGetOrientation** (function).

See Also **PrinterGetOrientation** (function).

Platform(s) Windows.

Platform Notes **Windows:** The default printer is determined by examining the device= line in the [windows] section of the win.ini file.

PrintFile (function)

Syntax `PrintFile(`*filename$*`)`

Description Prints the *filename$* using the application to which the file belongs.

Comments **PrintFile** returns an **Integer** indicating success or failure.

If an error occurs executing the associated application, then **PrintFile** generates a trappable runtime error, returning 0 for the result. Otherwise, **PrintFile** returns a value representing that application to the system. This value is suitable for calling the **AppActivate** statement.

Example

```
'This example asks the user for the name of a text file, then
'prints it.
Sub Main()
   f$ = OpenFilename$("Print Text File","Text Files:*.txt")
   If f$ <> "" Then
      rc% = PrintFile(f$)
      If rc% > 32 Then
         MsgBox "File is printing."
      End If
   End If
End Sub
```

See Also **Shell** (function).

Platform(s) Windows.

Platform Notes **Windows:** This function invokes the Windows 3.1 shell functions that cause an application to execute and print a file. The application executed by **PrintFile** depends on your system's file associations.

Private (statement)

Syntax `Private` *name* `[(`*subscripts*`)] [As` *type*`] [,`*name* `[(`*subscripts*`)] [As` *type*`]]...`

Description Declares a list of private variables and their corresponding types and sizes.

Comments Private variables are global to every **Sub** and **Function** within the currently executing script.

If a type-declaration character is used when specifying name (such as **%**, **@**, **&**, **$**, or **!**), the optional [**As** *type*] expression is not allowed. For example, the following are allowed:

```
Private foo As Integer
Private foo%
```

The *subscripts* parameter allows the declaration of arrays. This parameter uses the following syntax:

`[`*lower* `To]` *upper* `[,[`*lower* `To]` *upper*`]...`

The *lower* and *upper* parameters are integers specifying the lower and upper bounds of the array. If *lower* is not specified, then the lower bound as specified by **Option Base** is used (or 1 if no **Option Base** statement has been encountered). Up to 60 array dimensions are allowed.

The total size of an array (not counting space for strings) is limited to 64K.

Dynamic arrays are declared by not specifying any bounds:

```
Private a()
```

The *type* parameter specifies the type of the data item being declared. It can be any of the following data types: **String**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Object**, data object, built-in data type, or any user-defined data type.

If a variable is seen that has not been explicitly declared with either **Dim**, **Public**, or **Private**, then it will be implicitly declared local to the routine in which it is used.

Fixed-Length Strings

Fixed-length strings are declared by adding a length to the **String** type-declaration character:

```
Private name As String * length
```

where *length* is a literal number specifying the string's length.

Initial Values

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0
Long	0
Double	0.0
Single	0.0
Currency	0.0
Object	**Nothing**
Date	December 31, 1899 00:00:00
Boolean	**False**
Variant	**Empty**
String	"" (zero-length string)
User-defined type	Each element of the structure is given a default value, as described above.
Arrays	Each element of the array is given a default value, as described above.

Example See **Public** (statement).

See Also **Dim** (statement); **Redim** (statement); **Public** (statement); **Option Base** (statement).

Platform(s) All.

Public (statement)

Syntax `Public` *name* `[(`*subscripts*`)] [As` *type*`] [,`*name* `[(`*subscripts*`)] [As` *type*`]]...`

Description Declares a list of public variables and their corresponding types and sizes.

Comments Public variables are global to all **Sub**s and **Function**s in all scripts.

If a type-declaration character is used when specifying name (such as **%**, **@**, **&**, **$**, or **!**), the optional [**As** *type*] expression is not allowed. For example, the following are allowed:

```
Public foo As integer
Public foo%
```

The *subscripts* parameter allows the declaration of arrays. This parameter uses the following syntax:

`[`*lower* `To] ` *upper* `[,[`*lower* `To] ` *upper*`]...`

The *lower* and *upper* parameters are integers specifying the lower and upper bounds of the array. If *lower* is not specified, then the lower bound as specified by **Option Base** is used (or 1 if no **Option Base** statement has been encountered). Up to 60 array dimensions are allowed.

The total size of an array (not counting space for strings) is limited to 64K.

Dynamic arrays are declared by not specifying any bounds:

```
Public a()
```

The *type* parameter specifies the type of the data item being declared. It can be any of the following data types: **String**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Object**, data object, built-in data type, or any user-defined data type.

If a variable is seen that has not been explicitly declared with either **Dim**, **Public**, or **Private**, then it will be implicitly declared local to the routine in which it is used.

For compatibility, the keyword **Global** is also supported. It has the same meaning as **Public**.

Fixed-Length Strings

Fixed-length strings are declared by adding a length to the **String** type-declaration character:

`Public` *name* `As String * ` *length*

where *length* is a literal number specifying the string's length.

All declared variables are given initial values, as described in the following table:

Data Type	Initial Value
Integer	0

Data Type	Initial Value
Long	0
Double	0.0
Single	0.0
Currency	0.0
Date	December 31, 1899 00:00:00
Object	**Nothing**
Boolean	**False**
Variant	**Empty**
String	"" (zero-length string)
User-defined type	Each element of the structure is given a default value, as described above.
Arrays	Each element of the array is given a default value, as described above.

Sharing Variables

When sharing variables, you must ensure that the declarations of the shared variables are the same in each script that uses those variables. If the public variable being shared is a user-defined structure, then the structure definitions must be exactly the same.

Example

```
'This example uses a subroutine to calculate the area of ten
'circles and displays the result in a dialog box. The variables
'R and Ar are declared as Public variables so that they can be
'used in both Main and Area.
Const crlf = Chr$(13) + Chr$(10)
Public x#, ar#
Sub Area()
   ar# = (x# ^ 2) * Pi
End Sub
Sub Main()
   message = "The area of the ten circles are:" & crlf
   For x# = 1 To 10
      Area
      message = message & x# & ": " & ar# & Basic.Eoln$
   Next x#
   MsgBox message
End Sub
```

See Also **Dim** (statement); **Redim** (statement); **Private** (statement); **Option Base** (statement).

Platform(s) All.

PushButton (statement)

Syntax `PushButton` *x*,*y*,*width*,*height*,*title$* [,*.Identifier*]

Description Defines a push button within a dialog box template.

Comments Choosing a push button causes the dialog box to close (unless the dialog function redefines this behavior).

This statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

The **PushButton** statement accepts the following parameters:

Parameter	Description
x, y	**Integer** coordinates specifying the position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** coordinates specifying the dimensions of the control in dialog units.
title$	**String** containing the text that appears within the push button. This text may contain an ampersand character to denote an accelerator letter, such as "&Save" for Save.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**).

If a push button is the default button, it can be selected by pressing Enter on a nonbutton control.

A dialog box template must contain at least one **OKButton**, **CancelButton**, or **PushButton** statement (otherwise, the dialog box cannot be dismissed).

Example

```
'This example creates a bunch of push buttons and displays which
'button was pushed.
Sub Main()
   Begin Dialog ButtonTemplate 17,33,104,84,"Buttons"
      OKButton 8,4,40,14,.OK
      CancelButton 8,24,40,14,.Cancel
      PushButton 8,44,40,14,"1",.Button1
      PushButton 8,64,40,14,"2",.Button2
      PushButton 56,4,40,14,"3",.Button3
      PushButton 56,24,40,14,"4",.Button4
      PushButton 56,44,40,14,"5",.Button5
      PushButton 56,64,40,14,"6",.Button6
   End Dialog
   Dim ButtonDialog As ButtonTemplate
   WhichButton% = Dialog(ButtonDialog)
   MsgBox "You pushed button " & WhichButton%
```

```
End Sub
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **Text** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32, OS/2:** On Windows, Win32, and OS/2 platforms, accelerators are underlined, and the accelerator combination Alt+*letter* is used.

Macintosh: On the Macintosh, accelerators are normal in appearance, and the accelerator combination Command+*letter* is used.

Put (statement)

Syntax `Put [#]`*filenumber*`, [`*recordnumber*`], `*variable*

Description Writes data from the specified variable to a **Random** or **Binary** file.

Comments The **Put** statement accepts the following parameters:

Parameter	Description
filenumber	**Integer** representing the file to be written to. This is the same value as returned by the **Open** statement.
recordnumber	**Long** specifying which record is to be written to the file. For **Binary** files, this number represents the first byte to be written starting with the beginning of the file (the first byte is 1). For **Random** files, this number represents the record number starting with the beginning of the file (the first record is 1). This value ranges from 1 to 2147483647. If the *recordnumber* parameter is omitted, the next record is written to the file (if no records have been written yet, then the first record in the file is written). When *recordnumber* is omitted, the commas must still appear, as in the following example: `Put #1,,recvar` If *recordlength* is specified, it overrides any previous change in file position specified with the **Seek** statement.

The *variable* parameter is the name of any variable of any of the following types:

VariableType	File Storage Description
Integer	2 bytes are written to the file.
Long	4 bytes are written to the file.
String (variable-length)	In **Binary** files, variable-length strings are written by first determining the specified string variable's length, then writing that many bytes to a file. In **Random** files, variable-length strings are written by first writing a 2-byte length, then writing that many characters to the file.
String (fixed-length)	Fixed-length strings are written to **Random** and **Binary** files in the same way: the number of characters equal to the string's declared length are written.
Double	8 bytes are written to the file (IEEE format),
Single	4 bytes are written to the file (IEEE format).
Date	8 bytes are written to the file (IEEE double format).
Boolean	2 bytes are written to the file (either –1 for **True** or 0 for **False**).
Variant	A 2-byte **VarType** is written to the file followed by the data as described above. With variants of type 10 (user-defined errors), the 2-byte **VarType** is followed by a 4-byte error value (the low word containing the error value and the high word containing additional bytes of information). The exception is with strings, which are always preceded by a 2-byte string length.
User-defined types	Each member of a user-defined data type is written individually. In **Binary** files, variable-length strings within user-defined types are written by first writing a 2-byte length followed by the string's content. This storage is different than variable-length strings outside of user-defined types. When writing user-defined types, the record length must be greater than or equal to the combined size of each element within the data type.
Arrays	Arrays cannot be written to a file using the **Put** statement.

VariableType	File Storage Description
Objects	Object variables cannot be written to a file using the **Put** statement.

With **Random** files, a runtime error will occur if the length of the data being written exceeds the record length (specified as the *reclen* parameter with the **Open** statement). If the length of the data being written is less than the record length, the entire record is written along with padding (whatever data happens to be in the I/O buffer at that time). With **Binary** files, the data elements are written contiguously: they are never separated with padding.

Example

```
'This example opens a file for random write, then writes ten
'records into the file with the values 10-50. Then the file is
'closed and reopened in random mode for read, and the records
'are read with the Get statement. The result is displayed in a
'dialog box.
Sub Main()
   Open "test.dat" For Random Access Write As #1
   For x = 1 To 10
      r% = x * 10
      Put #1,x,r%
   Next x
   Close
   Open "test.dat" For Random Access Read As #1
   For x = 1 To 10
      Get #1,x,r%
      message = message & "Record " & x & " is: " & r% _
         & Basic.Eoln$
   Next x
   MsgBox message
   Close
   Kill "test.dat"
End Sub
```

See Also **Open** (statement); **Put** (statement); **Write#** (statement); **Print#** (statement).

Platform(s) All.

Pv (function)

Syntax `Pv(`*rate*`, `*nper*`, `*pmt*`, `*fv*`, `*due*`)`

Description Calculates the present value of an annuity based on future periodic fixed payments and a constant rate of interest.

Comments The **Pv** function requires the following named parameters:

Named Parameter	Description
rate	**Double** representing the interest rate per period. When used with monthly payments, be sure to normalize annual percentage rates by dividing them by 12.
nper	**Double** representing the total number of payments in the annuity.
pmt	**Double** representing the amount of each payment per period.
fv	**Double** representing the future value of the annuity after the last payment has been made. In the case of a loan, the future value would be 0.
due	**Integer** indicating when the payments are due for each payment period. A 0 specifies payment at the end of each period, whereas a 1 specifies payment at the start of each period.

The *rate* and *nper* parameters must be expressed in the same units. If *rate* is expressed in months, then *nper* must also be expressed in months.

Positive numbers represent cash received, whereas negative numbers represent cash paid out.

Example

```
'This example demonstrates the present value (the amount you'd
'have to pay now) for a $100,000 annuity that pays an annual
'income of $5,000 over 20 years at an annual interest rate of 10%.
Sub Main()
   pval = Pv(.1,20,-5000,100000,1)
   MsgBox "The present value is: " & Format(pval,"Currency")
End Sub
```

See Also **Fv** (function); **IRR** (function); **MIRR** (function); **Npv** (function).

Platform(s) All.

QueEmpty (statement)

Syntax

```
QueEmpty
```

Description Empties the current event queue.

Comments After this statement, **QueFlush** will do nothing.

Example

```
'This code begins a new queue, then drags a selection over a
'range of characters in Notepad.
```

```
Sub Main()
   AppActivate "Notepad"
   QueEmpty 'Make sure the queue is empty.
   QueMouseDn ebLeftButton,1440,1393
   QueMouseUp ebLeftButton,4147,2363
   QueFlush True
End Sub
```

Platform(s) Windows.

Platform Notes **Windows:** If a system modal dialog is invoked during queue playback, the queue playback is temporarily disabled. Queue playback will resume once the dialog has been dismissed. Hardware input is enabled during processing of the system modal dialog such that the dialog can be dismissed by the user. Otherwise, hardware input is enabled until playback is finished.

QueFlush (statement)

Syntax `QueFlush` *isSaveState*

Description Plays back events that are stored in the current event queue.

Comments After **QueFlush** is finished, the queue is empty.

If *isSaveState* is **True**, then **QueFlush** saves the state of the Caps Lock, Num Lock, Scroll Lock, and Insert and restores the state after the **QueFlush** is complete. If this parameter is **False**, these states are not restored.

The function does not return until the entire queue has been played.

Example

```
'This example pumps some keys into Notepad.
Sub Main()
   AppActivate "Notepad"
   QueKeys "This is a test{Enter}"
   QueFlush True        'Play back the queue.
End Sub
```

Platform(s) Windows.

Platform Notes **Windows:** The **QueFlush** statement uses the Windows journaling mechanism to replay the mouse and keyboard events stored in the queue. As a result, the mouse position may be changed. Furthermore, events can be played into any Windows application, including DOS applications running in a window.

QueKeyDn (statement)

Syntax `QueKeyDn` *KeyString$* `[,`*time*`]`

Description Appends key-down events for the specified keys to the end of the current event queue.

Comments The **QueKeyDn** statement accepts the following parameters:

Parameter	Description
KeyString$	**String** containing the keys to be sent. The format for *KeyString$* is described under the **SendKeys** statement.
time	**Integer** specifying the number of milliseconds devoted for the output of the entire *KeyString$* parameter. It must be within the following range: `0 <= time <= 32767` For example, if *time* is 5000 (5 seconds) and the *KeyString$* parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (or 0), the keys will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
'This example plays back a Ctrl + mouse click.
Sub Main()
   QueEmpty
   QueKeyDn "^"
   QueMouseClick ebLeftButton 1024,792
   QueKeyUp "^"
   QueFlush True
End Sub
```

See Also **DoKeys** (statement); **SendKeys** (statement); **QueKeys** (statement); **QueKeyUp** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueKeys (statement)

Syntax `QueKeys` *KeyString$* `[,`*time*`]`

Description Appends keystroke information to the current event queue.

Comments The **QueKeys** statement accepts the following parameters:

Parameter	Description
KeyString$	**String** containing the keys to be sent. The format for *KeyString$* is described under the **SendKeys** statement.

Parameter	Description
time	**Integer** specifying the number of milliseconds devoted for the output of the entire *KeyString$* parameter. It must be within the following range: `0 <= time <= 32767` For example, if *time* is 5000 (5 seconds) and the *KeyString$* parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (or 0), the keys will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
Sub Main()
   WinActivate "Notepad"
   QueEmpty
   QueKeys "This is a test.{Enter}This is on a new line.{Enter}"
   QueKeys "{Tab 3}This is indented with three tabs."
   QueKeys "Some special characters: {~}{^}{%}{+}~"
   QueKeys "Invoking the Find dialog.%Sf"'Alt+S,F
   QueFlush True
End Sub
```

See Also **DoKeys** (statement); **SendKeys** (statement); **QueKeyDn** (statement); **QueKeyUp** (statement); **QueFlush** (statement).

Platform(s) Windows.

Platform Notes **Windows:** Under Windows, you cannot send keystrokes to MS-DOS applications running in a window.

QueKeyUp (statement)

Syntax `QueKeyUp` *KeyString$* `[,`*time*`]`

Description Appends key-up events for the specified keys to the end of the current event queue.

Comments The **QueKeyUp** statement accepts the following parameters:

Parameter	Description
KeyString$	**String** containing the keys to be sent. The format for *KeyString$* is described under the **SendKeys** statement.

Parameter	Description
time	**Integer** specifying the number of milliseconds devoted for the output of the entire *KeyString$* parameter. It must be within the following range: `0 <= time <= 32767` For example, if *time* is 5000 (5 seconds) and the *KeyString$* parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (or 0), the keys will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example See **QueKeyDn** (statement).

See Also **DoKeys** (statement); **SendKeys** (statement); **QueKeys** (statement); **QueKeyDn** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseClick (statement)

Syntax `QueMouseClick` *button*`,`*x*`,`*y* `[,`*time*`]`

Description Adds a mouse click to the current event queue.

Comments The **QueMouseClick** statement takes the following parameters:

Parameter	Description
button	**Integer** specifying which mouse button to click: **ebLeftButton**Click the left mouse button. **ebRightButton**Click the right mouse button.
x, *y*	**Integer** coordinates, in twips, where the mouse click is to be recorded.
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse click will play back at full speed.

A mouse click consists of a mouse button down at position *x*, *y*, immediately followed by a mouse button up.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
'This example activates Notepad and invokes the Find dialog box.
```

```
'It then uses the QueMouseClick command to click the Cancel
'button.
Sub Main()
   AppActivate "Notepad"'Activate Notepad.
   QueKeys "%Sf"'Invoke the Find dialog box.
   QueFlush True'Play this back (allow dialog box to open).
   QueSetRelativeWindow'Set mouse relative to Find dialog box.
   QueMouseClick ebLeftButton,7059,1486'Click the Cancel button.
   QueFlush True'Play back the queue.
End Sub
```

See Also **QueMouseDn** (statement); **QueMouseUp** (statement); **QueMouseDblClk** (statement); **QueMouseDblDn** (statement); **QueMouseMove** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseDblClk (statement)

Syntax `QueMouseDblClk` *button*`,`*x*`,`*y* `[,`*time*`]`

Description Adds a mouse double click to the current event queue.

Comments The **QueMouseDblClk** statement takes the following parameters:

Parameter	Description
button	**Integer** specifying which mouse button to double-click: **ebLeftButton**Double-click the left mouse button. **ebRightButton**Double-click the right mouse button.
x, *y*	**Integer** coordinates, in twips, where the mouse double click is to be recorded.
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse double click will play back at full speed.

A mouse double click consists of a mouse down/up/down/up at position *x*, *y*. The events are queued in such a way that a double click is registered during queue playback.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
'This example double-clicks the left mouse button.
QueMouseDblClk ebLeftButton,344,360
```

See Also **QueMouseClick** (statement); **QueMouseDn** (statement); **QueMouseUp** (statement); **QueMouseDblDn** (statement); **QueMouseMove** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseDblDn (statement)

Syntax `QueMouseDblDn` *button*`,` *x*`,` *y* `[,`*time*`]`

Description Adds a mouse double down to the end of the current event queue.

Comments The **QueMouseDblDn** statement takes the following parameters:

Parameter	Description
button	**Integer** specifying which mouse button to press: **ebLeftButton**Press the left mouse button. **ebRightButton**Press the right mouse button.
x, *y*	**Integer** coordinates, in twips, where the mouse double down is to be recorded.
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse double down will play back at full speed.

This statement adds a mouse double down to the current event queue. A double down consists of a mouse down/up/down at position *x*, *y*.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
'This example double-clicks a word, then drags it to a new
'location.
Sub Main()
   QueFlush                  'Start with empty queue.
   QueMouseDblDn ebLeftButton,356,4931
   QueMouseMove 600,4931'Drag to new spot.
   QueMouseUp ebLeftButton'Now release the mouse.
   QueFlush True             'Play back the queue.
End Sub
```

See Also **QueMouseClick** (statement); **QueMouseDn** (statement); **QueMouseUp** (statement); **QueMouseDblClk** (statement); **QueMouseMove** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseDn (statement)

Syntax `QueMouseDn` *button*`,`*x*`,`*y* `[,`*time*`]`

Description Adds a mouse down to the current event queue.

Comments The **QueMouseDn** statement takes the following parameters:

Parameter	Description
button	**Integer** specifying which mouse button to press: **ebLeftButton**Press the left mouse button. **ebRightButton**Press the right mouse button.
x, *y*	**Integer** coordinates, in twips, where the mouse down is to be recorded.
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse down will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example See **QueEmpty** (statement).

See Also **QueMouseClick** (statement); **QueMouseUp** (statement); **QueMouseDblClk** (statement); **QueMouseDblDn** (statement); **QueMouseMove** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseMove (statement)

Syntax `QueMouseMove` *x*`,`*y* `[,`*time*`]`

Description Adds a mouse move to the current event queue.

Comments The **QueMouseMove** statement takes the following parameters:

Parameter	Description
x, *y*	**Integer** coordinates, in twips, where the mouse is to be moved.
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse move will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example See **QueMouseDblDn** (statement).

See Also **QueMouseClick** (statement); **QueMouseDn** (statement); **QueMouseUp** (statement); **QueMouseDblClk** (statement); **QueMouseDblDn** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseMoveBatch (statement)

Syntax `QueMouseMoveBatch` *ManyMoves*`$`

Description Adds a series of mouse-move events to the current event queue.

Comments The *ManyMoves$* parameter is a string containing positional and timing information in the following format:

x,*y*,*time* [,*x*,*y*,*time*]...

The *x* and *y* parameters specify a mouse position in twips. The *time* parameter specifies the delay in milliseconds between the current mouse move and the previous event in the queue. If *time* is 0, then the mouse move will play back as fast as possible.

The **QueMouseMoveBatch** command should be used in place of a series of **QueMouseMove** statements to reduce the number of lines in your script. A further advantage is that, since the mouse-move information is contained within a literal string, the storage for the data is placed in the constant segment instead of the code segment, reducing the size of the code.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
'This example activates PaintBrush, then paints the word "Hi".
Sub Main()
   AppActivate "Paintbrush"
   AppMaximize
   QueMouseDn ebLeftButton,2175,3412
   QueMouseMoveBatch _
   "2488,3224,0,2833,2786,0,3114,2347,0,3208,2160,0,3240,2097,0"
   QueMouseMoveBatch _
   "3255,2034,0,3255,1987,0,3255,1956,0,3255,1940,0,3224,1956,0"
   QueMouseMoveBatch _
   "3193,1987,0,3114,2019,0,3036,2066,0,3005,2113,0,2973,2175,0"
   QueMouseMoveBatch _
   "2942,2332,0,2926,2394,0,2926,2582,0,2911,2739,0,2911,2801,0"
   QueMouseMoveBatch _
   "2911,2958,0,2911,3020,0,2911,3052,0,2911,3083,0,2911,3114,0"
   QueMouseMoveBatch _
   "2911,3130,0,2895,3161,0,2895,3193,0,2895,3208,0,2895,3193,0"
```

```
    QueMouseMoveBatch _
    "2895,3146,0,2911,3083,0,2926,3020,0,2942,2958,0,2973,2895,0"
    QueMouseMoveBatch _
    "3005,2848,0,3020,2817,0,3036,2801,0,3052,2770,0,3083,2770,0"
    QueMouseMoveBatch _
    "3114,2754,0,3130,2754,0,3146,2770,0,3161,2786,0,3161,2848,0"
    QueMouseMoveBatch _
    "3193,3005,0,3193,3193,0,3208,3255,0,3224,3318,0,3240,3349,0"
    QueMouseMoveBatch _
    "3255,3349,0,3286,3318,0,3380,3271,0,3474,3208,0,3553,3052,0"
    QueMouseMoveBatch _
    "3584,2895,0,3615,2739,0,3631,2692,0,3631,2645,0,3646,2645,0"
    QueMouseMoveBatch _
    "3646,2660,0,3646,2723,0,3646,2880,0,3662,2942,0,3693,2989,0"
    QueMouseMoveBatch _
    "3709,3005,0,3725,3005,0,3756,2989,0,3787,2973,0"
    QueMouseUp ebLeftButton,3787,2973
    QueMouseDn ebLeftButton,3678,2535
    QueMouseMove 3678,2520
    QueMouseMove 3678,2535
    QueMouseUp ebLeftButton,3678,2535
    QueFlush True
End Sub
```

See Also **QueMouseClick** (statement); **QueMouseDn** (statement); **QueMouseUp** (statement); **QueMouseDblClk** (statement); **QueMouseDblDn** (statement); **QueMouseMove** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueMouseUp (statement)

Syntax `QueMouseUp` *button*`,`*x*`,`*y* `[,`*time*`]`

Description Adds a mouse up to the current event queue.

Comments The **QueMouseUp** statement takes the following parameters:

Parameter	Description
button	**Integer** specifying the mouse button to be released: **ebLeftButton**Release the left mouse button. **ebRightButton**Release the right mouse button.
x, *y*	**Integer** coordinates, in twips, where the mouse button is to be released.

Parameter	Description
time	**Integer** specifying the delay in milliseconds between this event and the previous event in the queue. If this parameter is omitted (or 0), the mouse up will play back at full speed.

The **QueFlush** command is used to play back the events stored in the current event queue.

Example See **QueEmpty** (statement).

See Also **QueMouseClick** (statement); **QueMouseDn** (statement); **QueMouseDblClk** (statement); **QueMouseDblDn** (statement); **QueMouseMove** (statement); **QueMouseMoveBatch** (statement); **QueFlush** (statement).

Platform(s) Windows.

QueSetRelativeWindow (statement)

Syntax `QueSetRelativeWindow [`*window_object*`]`

Description Forces all subsequent **Que***X* commands to adjust the mouse positions relative to the specified window.

Comments The *window_object* parameter is an object of type HWND. If *window_object* is **Nothing** or omitted, then the window with the focus is used (i.e., the active window).

The **QueFlush** command is used to play back the events stored in the current event queue.

Example

```
Sub Main()
   'Adjust mouse coordinates relative to Notepad.
   Dim a As HWND
   Set a = WinFind("Notepad")
   QueSetRelativeWindow a
End Sub
```

Platform(s) Windows.

Random (function)

Syntax `Random(`*min*`,`*max*`)`

Description Returns a **Long** value greater than or equal to *min* and less than or equal to *max*.

Comments Both the *min* and *max* parameters are rounded to **Long**. A runtime error is generated if *min* is greater than *max*.

Example

```
'This example uses the random number generator to generate ten
'lottery numbers.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Randomize         'Start with new random seed.
   For x = 1 To 10
      y = Random(0,100)'Generate numbers.
      message = message & y & crlf
   Next x
   MsgBox "Ten numbers for the lottery: " & crlf & message
End Sub
```

See Also **Randomize** (statement); **Random** (function).

Platform(s) All.

Randomize (statement)

Syntax `Randomize [`*number*`]`

Description Initializes the random number generator with a new seed.

Comments If *number* is not specified, then the current value of the system clock is used.

Example

```
'This example sets the randomize seed to a random number between
'100 and 1000, then generates ten random numbers for the lottery.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Randomize         'Start with new random seed.
   For x = 1 To 10
      y = Random(0,100)'Generate numbers.
      message = message + Str(y) + crlf
   Next x
   MsgBox "Ten numbers for the lottery: " & crlf & message
End Sub
```

See Also **Random** (function); **Rnd** (function).

Platform(s) All.

Rate (function)

Syntax `Rate(`*nper*`, `*pmt*`, `*pv*`, `*fv*`, `*due*`, `*guess*`)`

Description Returns the rate of interest for each period of an annuity.

Comments An annuity is a series of fixed payments made to an insurance company or other investment company over a period of time. Examples of annuities are mortgages and monthly savings plans.

The **Rate** function requires the following named parameters:

Named Parameter	Description
nper	**Double** representing the total number of payments in the annuity.
pmt	**Double** representing the amount of each payment per period.
pv	**Double** representing the present value of your annuity. In a loan situation, the present value would be the amount of the loan.
fv	**Double** representing the future value of the annuity after the last payment has been made. In the case of a loan, the future value would be zero.
due	**Integer** specifying when the payments are due for each payment period. A 0 indicates payment at the end of each period, whereas a 1 indicates payment at the start of each period.
guess	**Double** specifying a guess as to the value the **Rate** function will return. The most common guess is .1 (10 percent).

Positive numbers represent cash received, whereas negative values represent cash paid out.

The value of **Rate** is found by iteration. It starts with the value of *guess* and cycles through the calculation adjusting *guess* until the result is accurate within 0.00001 percent. After 20 tries, if a result cannot be found, **Rate** fails, and the user must pick a better guess.

Example

```
'This example calculates the rate of interest necessary to save
'$8,000 by paying $200 each year for 48 years. The guess rate
'is 10%.
Sub Main()
   r# = Rate(48,-200,8000,0,1,.1)
   MsgBox "The rate required is: " & Format(r#,"Percent")
End Sub
```

See Also **IPmt** (function); **NPer** (function); **Pmt** (function); **PPmt** (function).

Platform(s) All.

ReadIni$ (function)

Syntax `ReadIni$(`*section$*`,`*item$*`[,`*filename$*`])`

Description Returns a **String** containing the specified item from an ini file.

Comments The **ReadIni$** function takes the following parameters:

Parameter	Description
section$	**String** specifying the section that contains the desired variable, such as "windows". Section names are specified without the enclosing brackets.
item$	**String** specifying the item whose value is to be retrieved.
filename$	**String** containing the name of the ini file to read.

The maximum length of a string returned by this function is 4096 characters.

See Also **WriteIni** (statement); **ReadIniSection** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows and Win32, if the name of the ini file is not specified, then win.ini is assumed.

If the *filename$* parameter does not include a path, then this statement looks for ini files in the Windows directory.

ReadIniSection (statement)

Syntax `ReadIniSection` *section$*`,`*ArrayOfItems()*`[,`*filename$*`]`

Description Fills an array with the item names from a given section of the specified ini file.

Comments The **ReadIniSection** statement takes the following parameters:

Parameter	Description
section$	**String** specifying the section that contains the desired variables, such as "windows". Section names are specified without the enclosing brackets.
ArrayOfItems()	Specifies either a zero- or a one-dimensioned array of strings or variants. The array can be either dynamic or fixed. If *ArrayOfItems()* is dynamic, then it will be redimensioned to exactly hold the new number of elements. If there are no elements, then the array will be redimensioned to contain no dimensions. You can use the **LBound**, **UBound**, and **ArrayDims** functions to determine the number and size of the new array's dimensions.

Parameter	Description
	If the array is fixed, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are initialized to zero-length strings (for **String** arrays) or **Empty** (for **Variant** arrays). A runtime error results if the array is too small to hold the new elements.
filename$	**String** containing the name of an ini file.

On return, the *ArrayOfItems()* parameter will contain one array element for each variable in the specified ini section. The maximum combined length of all the entry names returned by this function is limited to 32K.

Example

```
Sub Main()
   Dim items() As String
   ReadIniSection "windows",items$
   r% = SelectBox("INI Items",,items$)
End Sub
```

See Also **ReadIni$** (function); **WriteIni** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows and Win32, if the name of the ini file is not specified, then win.ini is assumed.

If the *filename$* parameter does not include a path, then this statement looks for ini files in the Windows directory.

Redim (statement)

Syntax `Redim [Preserve] variablename ([subscriptRange]) [As type],...`

Description Redimensions an array, specifying a new upper and lower bound for each dimension of the array.

Comments The *variablename* parameter specifies the name of an existing array (previously declared using the **Dim** statement) or the name of a new array variable. If the array variable already exists, then it must previously have been declared with the **Dim** statement with no dimensions, as shown in the following example:

```
Dim a$()    'Dynamic array of strings (no dimensions yet)
```

Dynamic arrays can be redimensioned any number of times.

The *subscriptRange* parameter specifies the new upper and lower bounds for each dimension of the array using the following syntax:

```
[lower To] upper [,[lower To] upper]...
```

If *subscriptRange* is not specified, then the array is redimensioned to have no elements.

If *lower* is not specified, then 0 is used (or the value set using the **Option Base** statement). A runtime error is generated if *lower* is less than *upper*. Array dimensions must be within the following range:

```
-32768 <= lower <= upper <= 32767
```

The *type* parameter can be used to specify the array element type. Arrays can be declared using any fundamental data type, user-defined data types, and objects.

Redimensioning an array erases all elements of that array unless the **Preserve** keyword is specified. When this keyword is specified, existing data in the array is preserved where possible. If the number of elements in an array dimension is increased, the new elements are initialized to 0 (or empty string). If the number of elements in an array dimension is decreased, then the extra elements will be deleted. If the **Preserve** keyword is specified, then the number of dimensions of the array being redimensioned must either be zero or the same as the new number of dimensions.

Example

```
'This example uses the FileList statement to redim an array and
'fill it with filename strings. A new array is then redimmed to
'hold the number of elements found by FileList, and the FileList
'array is copied into it and partially displayed.
Sub Main()
   Dim fl$()
   FileList fl$,"*.*"
   count = Ubound(fl$)
   Redim nl$(Lbound(fl$) To Ubound(fl$))
   For x = 1 to count
      nl$(x) = fl(x)
   Next x
   MsgBox "The last element of the new array is: " & nl$(count)
End Sub
```

See Also **Dim** (statement); **Public** (statement); **Private** (statement); **ArrayDims** (function); **LBound** (function); **UBound** (function).

Platform(s) All.

Rem (statement)

Syntax `Rem` *text*

Description Causes the compiler to skip all characters on that line.

Example

```
Sub Main()
   Rem This is a line of comments that serves to illustrate the
   Rem workings of the code. You can insert comments to make it
   Rem more readable and maintainable in the future.
```

```
End Sub
```

See Also ' (keyword); Comments (topic).

Platform(s) All.

Reset (statement)

Syntax `Reset`

Description Closes all open files, writing out all I/O buffers.

Example

```
'This example opens a file for output, closes it with the Reset
'statement, then deletes it with the Kill statement.
Sub Main()
   Open "test.dat" for Output Access Write as # 1
   Reset
   Kill "test.dat"
   If FileExists("test.dat") Then
      MsgBox "The file was not deleted."
   Else
      MsgBox "The file was deleted."
   End If
End Sub
```

See Also **Close** (statement); **Open** (statement).

Platform(s) All.

Resume (statement)

Syntax `Resume {[0] | Next |` *label*`}`

Description Ends an error handler and continues execution.

Comments The form **Resume 0** (or simply **Resume** by itself) causes execution to continue with the statement that caused the error.

The form **Resume Next** causes execution to continue with the statement following the statement that caused the error.

The form **Resume** *label* causes execution to continue at the specified label.

The **Resume** statement resets the error state. This means that, after executing this statement, new errors can be generated and trapped as normal.

Example

```
'This example accepts two integers from the user and attempts
'to multiply the numbers together. If either number is larger
'than an integer, the program processes an error routine and
```

```
'then continues program execution at a specific section using
'"Resume <label>". Another error trap is then set using "Resume
'Next". The new error trap will clear any previous error
'branching and also "tell" the program to continue execution of
'the program even if an error is encountered.
Sub Main()
   Dim a%, b%, x%
Again:
   On Error Goto Overflow
   a% = InputBox("Enter 1st integer to multiply","Enter Number")
   b% = InputBox("Enter 2nd integer to multiply","Enter Number")
   On Error Resume Next    'Continue program execution at next
   x% = a% * b%            'line if an error occurs.
   if err = 0 then
      MsgBox x%
   else
      Msgbox a% & " * " & b% & " cause an overflow!"
   end if
   Exit Sub
Overflow:            'Error handler.
   MsgBox "You've entered a noninteger value. Try again!"
   Resume Again
End Sub
```

See Also Error Handling (topic); **On Error** (statement).

Platform(s) All.

Return (statement)

Syntax `Return`

Description Transfers execution control to the statement following the most recent **GoSub**.

Comments A runtime error results if a **Return** statement is encountered without a corresponding **GoSub** statement.

Example

```
'This example calls a subroutine and then returns execution to
'the Main routine by the Return statement.
Sub Main()
   GoSub SubTrue
   MsgBox "The Main routine continues here."
   Exit Sub
SubTrue:
   MsgBox "This message is generated in the subroutine."
   Return
   Exit Sub
End Sub
```

See Also **GoSub** (statement).

Platform(s) All.

Right, Right$, RightB, RightB$ (functions)

Syntax

```
Right[$](string, length)
RightB[$](string, length)
```

Description Returns the rightmost *length* characters (for **Right** and **Right$**) or bytes (for **RightB** and **RightB$**) from a specified string.

Comments The **Right$** and **RightB$** functions return a **String**, whereas the **Right** and **RightB** functions return a **String** variant.

These functions take the following named parameters:

Named Parameter	Description
string	**String** from which characters are returned. A runtime error is generated if *string* is **Null**.
length	**Integer** specifying the number of characters or bytes to return. If *length* is greater than or equal to the length of the string, then the entire string is returned. If *length* is 0, then a zero-length string is returned.

The **RightB** and **RightB$** functions are used to return byte data from strings containing byte data.

Example

```
'This example shows the Right$ function used in a routine to
'change uppercase names to lowercase with an uppercase first
'letter.
Sub Main()
   lname$ = "WILLIAMS"
   x = Len(lname$)
   rest$ = Right$(lname$,x - 1)
   fl$ =  Left$(lname$,1)
   lname$ = fl$ & LCase$(rest$)
   MsgBox "The converted name is: " & lname$
End Sub
```

See Also **Left**, **Left$, LeftB**, **LeftB$** (functions).

Platform(s) All.

RmDir (statement)

Syntax `RmDir` *path*

Description Removes the directory specified by the **String** contained in *path*.

Comments

Removing the Current Directory

On platforms that support drive letters, removing a directory that is the current directory on that drive causes unpredictable side effects. For example, consider the following statements:

```
MkDir "Z:\JUNK"
ChDir "Z:\JUNK"
RmDir "Z:\JUNK"
```

If this code is run under Windows and drive Z is a network drive, then some networks will delete the directory and unmap the drive without generating a script error. If drive Z is a local drive, the directory will not be deleted, nor will the script receive an error.

Different platforms and file systems exhibit similar strange behavior in these cases.

Example

```
'This routine creates a directory and then deletes it with RmDir.
Sub Main()
   On Error Goto ErrMake
   MkDir("test01")
   On Error Goto ErrRemove
   RmDir("test01")
ErrMake:
   MsgBox "The directory could not be created."
   Exit Sub
ErrRemove:
   MsgBox "The directory could not be removed."
   Exit Sub
End Sub
```

See Also **ChDir** (statement); **ChDrive** (statement); **CurDir**, **CurDir$** (functions); **Dir**, **Dir$** (functions); **MkDir** (statement).

Platform(s) All.

Platform Notes **Windows:** Under Windows, this command behaves the same as the DOS "rd" command.

Rnd (function)

Syntax `Rnd[(`*number*`)]`

Description Returns a random **Single** number between 0 and 1.

Comments If *number* is omitted, the next random number is returned. Otherwise, the *number* parameter has the following meaning:

If	Then
number < 0	Always returns the same number.
number = 0	Returns the last number generated.
number > 0	Returns the next random number.

Example

```
'This routine generates a list of random numbers and displays
'them.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   For x = -1 To 8
      y! = Rnd(1) * 100
      message = message & x & "  : " & y! & crlf
   Next x
   MsgBox message & "Last form: " & Rnd
End Sub
```

See Also **Randomize** (statement); **Random** (function).

Platform(s) All.

RSet (statement)

Syntax `RSet` *destvariable* `=` *source*

Description Copies the source string *source* into the destination string *destvariable*.

Comments If *source* is shorter in length than *destvariable*, then the string is right-aligned within *destvariable* and the remaining characters are padded with spaces. If *source* is longer in length than *destvariable*, then *source* is truncated, copying only the leftmost number of characters that will fit in *destvariable*. A runtime error is generated if *source* is **Null**.

The *destvariable* parameter specifies a **String** or **Variant** variable. If *destvariable* is a **Variant** containing **Empty**, then no characters are copied. If *destvariable* is not convertible to a **String**, then a runtime error occurs. A runtime error results if *destvariable* is **Null**.

Example

```
'This example replaces a 40-character string of asterisks (*)
'with an RSet and LSet string and then displays the result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim message,tmpstr$
   tmpstr$ = String$(40, "*")
   message = "Here are two strings that have been right-" & crlf
```

```
   message = message & "and left-justified in a " _
      & "40-character string."
   message = message & crlf & crlf
   RSet tmpstr$ = "Right->"
   message = message & tmpstr$ & crlf
   LSet tmpstr$ = "<-Left"
   message = message & tmpstr$ & crlf
   MsgBox message
End Sub
```

See Also **LSet** (statement).

Platform(s) All.

RTrim, RTrim$ (functions)

See **Trim**, **Trim$**, **LTrim**, **LTrim$**, **RTrim**, **RTrim$** (functions).

SaveFileName$ (function)

Syntax `SaveFileName$[([title$ [,[extensions$] [helpfile,context]]])]`

Description Displays a dialog box that prompts the user to select from a list of files and returns a **String** containing the full path of the selected file.

Comments The **SaveFileName$** function accepts the following parameters:

Parameter	Description
title$	**String** containing the title that appears on the dialog box's caption. If this string is omitted, then "Save As" is used.
extensions$	**String** containing the available file types. Its format depends on the platform on which BasicScript is running. If this string is omitted, then all files are used.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

The **SaveFileName$** function returns a full pathname of the file that the user selects. A zero-length string is returned if the user selects Cancel. If the file already exists, then the user is prompted to overwrite it.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 key on most platforms). Invoking help does not remove the dialog.

Example

```
'This example creates a save dialog box, giving the user the
'ability to save to several different file types.
Sub Main()
      e$ = "All Files:*.BMP,*.WMF;Bitmaps:*.BMP;Metafiles:*.WMF"
      f$ = SaveFileName$("Save Picture",e$)
      If Not f$ = "" Then
         MsgBox "User choose to save file as: " + f$
      Else
         MsgBox "User canceled."
      End If
End Sub
```

See Also **MsgBox** (statement); **AskBox**, **AskBox$** (functions); **AskPassword**, **AskPassword$** (functions); **InputBox**, **InputBox$** (functions); **OpenFileName$** (function); **SelectBox** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32:** Under Windows andWin32 , the *extensions$* parameter must be in the following format:

description:*ext*[,*ext*][;*description*:*ext*[,*ext*]]...

Placeholder	Description
description	Specifies the grouping of files for the user, such as **All Files**.
ext	Specifies a valid file extension, such as ***.BAT** or ***.?F?**.

For example, the following are valid *extensions$* specifications:

```
"All Files:*"
"Documents:*.TXT,*.DOC"
"All Files:*;Documents:*.TXT,*.DOC"
```

OS/2: Under OS/2, the *extensions$* parameter is a comma-delimited list of extended attribute names. An entry for **<All Files>** will always appear in the File Types list, regardless of the contents of the *extensions$* parameter. For example, the following is a valid *extensions$* specification:

```
"OS/2 Command File,Plain Text"
```

Macintosh: On the Macintosh, the *extensions$* parameter contains a comma-separated list of four-character file types. For example:

```
"TEXT,XLS4,MSWD"
```

On the Macintosh, the *title$* parameter is ignored.

SaveSetting (statement)

Syntax `SaveSetting` *appname*, *section*, *key*, *setting*

Description Saves the value of the specified key in the system registry. The following table describes the named parameters to the **SaveSetting** statement:

Named Parameter	Description
appname	**String** expression indicating the name of the application whose setting will be modified.
section	**String** expression indicating the name of the section whose setting will be modified.
key	**String** expression indicating the name of the setting to be modified.
setting	The value assigned to *key*.

Example

```
'The following example adds two entries to the Windows registry
'if run under Win32 or to NEWAPP.INI on other platforms,
```

```
'using the SaveSetting statement. It then uses DeleteSetting
'to remove these entries.
Sub Main()
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Height", setting := 200
   SaveSetting appname := "NewApp", section := "Startup", _
      key := "Width", setting := 320
   DeleteSetting "NewApp"     'Remove NewApp key from registry
End Sub
```

See Also **GetAllSettings** (function); **DeleteSetting** (statement); **GetSetting** (function).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Win32:** Under Win32, this statement operates on the system registry. All settings are saved to the following entry in the system registry:

```
HKEY_CURRENT_USER\Software\BasicScript Program
Settings\appname\section\key
```

On this platform, the *appname* parameter is not optional.

Windows, OS/2: Settings are stored in INI files. The name of the INI file is specified by *appname*. If *appname* is omitted, then this command operates on the WIN.INI file. For example, to change the **Language** setting from the **intl** section of the WIN.INI file, you could use the following statement:

```
s$ = SaveSetting(,"intl","sLanguage","eng")
```

Screen.DlgBaseUnitsX (property)

Syntax

```
Screen.DlgBaseUnitsX
```

Description Returns an **Integer** used to convert horizontal pixels to and from dialog units.

Comments The number returned depends on the name and size of the font used to display dialog boxes.

To convert from pixels to dialog units in the horizontal direction:

```
((XPixels * 4) + (Screen.DlgBaseUnitsX - 1)) /
Screen.DlgBaseUnitsX
```

To convert from dialog units to pixels in the horizontal direction:

```
(XDlgUnits * Screen.DlgBaseUnitsX) / 4
```

Example

```
'This example converts the screen width from pixels to dialog
'units.
Sub Main()
   XPixels = Screen.Width
   conv% = Screen.DlgBaseUnitsX
   XDlgUnits = (XPixels * 4) + (conv% -1) / conv%
```

```
   MsgBox "The screen width is " & XDlgUnits & " dialog units."
End Sub
```

See Also **Screen.DlgBaseUnitsY** (property).

Platform(s) Windows, Win32.

Screen.DlgBaseUnitsY (property)

Syntax

```
Screen.DlgBaseUnitsY
```

Description Returns an **Integer** used to convert vertical pixels to and from dialog units.

Comments The number returned depends on the name and size of the font used to display dialog boxes.

To convert from pixels to dialog units in the vertical direction:

```
(YPixels * 8) + (Screen.DlgBaseUnitsY - 1) /
Screen.DlgBaseUnitsY
```

To convert from dialog units to pixels in the vertical direction:

```
(YDlgUnits * Screen.DlgBaseUnitsY) / 8
```

Example

```
'This example converts the screen width from pixels to dialog
'units.
Sub Main()
   YPixels = Screen.Height
   conv% = Screen.DlgBaseUnitsY
   YDlgUnits = (YPixels * 8) + (conv% -1) / conv%
   MsgBox "The screen width is " & YDlgUnits & " dialog units."
End Sub
```

See Also **Screen.DlgBaseUnitsX** (property).

Platform(s) Windows, Win32.

Screen.Height (property)

Syntax

```
Screen.Height
```

Description Returns the height of the screen in pixels as an **Integer**.

Comments This property is used to retrieve the height of the screen in pixels. This value will differ depending on the display resolution.

This property is read-only.

Example

```
'This example displays the screen height in pixels.
Sub Main()
```

```
   MsgBox "The Screen height is " & Screen.Height & " pixels."
End Sub
```

See Also **Screen.Width** (property).

Platform(s) Windows, Win32.

Screen.TwipsPerPixelX (property)

Syntax `Screen.TwipsPerPixelX`

Description Returns an **Integer** representing the number of twips per pixel in the horizontal direction of the installed display driver.

Comments This property is read-only.

Example

```
'This example displays the number of twips across the screen
'horizontally.
Sub Main()
   XScreenTwips = Screen.Width * Screen.TwipsPerPixelX
   MsgBox "Total horizontal screen twips = " & XScreenTwips
End Sub
```

See Also **Screen.TwipsPerPixelY** (property).

Platform(s) Windows, Win32.

Screen.TwipsPerPixelY (property)

Syntax `Screen.TwipsPerPixelY`

Description Returns an **Integer** representing the number of twips per pixel in the vertical direction of the installed display driver.

Comments This property is read-only.

Example

```
'This example displays the number of twips across the screen
'vertically.
Sub Main()
   YScreenTwips = Screen.Height * Screen.TwipsPerPixelY
   MsgBox "Total vertical screen twips = " & YScreenTwips
End Sub
```

See Also **Screen.TwipsPerPixelX** (property).

Platform(s) Windows, Win32.

Screen.Width (property)

Syntax `Screen.Width`

Description Returns the width of the screen in pixels as an **Integer**.

Comments This property is used to retrieve the width of the screen in pixels. This value will differ depending on the display resolution.

This property is read-only.

Example
```
'This example displays the screen width in pixels.
Sub Main()
   MsgBox "The screen width is " & Screen.Width & " pixels."
End Sub
```

See Also **Screen.Height** (property).

Platform(s) Windows, Win32.

Second (function)

Syntax `Second(`*time*`)`

Description Returns the second of the day encoded in the specified *time* parameter.

Comments The value returned is an **Integer** between 0 and 59 inclusive.

The *time* parameter is any expression that converts to a **Date**.

Example
```
'This example takes the current time; extracts the hour, minute,
'and second; and displays them as the current time.
Sub Main()
   xt# = TimeValue(Time$())
   xh# = Hour(xt#)
   xm# = Minute(xt#)
   xs# = Second(xt#)
   Msgbox "The current time is: " & CStr(xh#) & ":" & CStr(xm#) _
      & ":" & CStr(xs#)
End Sub
```

See Also **Day** (function); **Minute** (function); **Month** (function); **Year** (function); **Hour** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

Seek (function)

Syntax `Seek(`*filenumber*`)`

Description Returns the position of the file pointer in a file relative to the beginning of the file.

Comments The *filenumber* parameter is a number that BasicScript uses to refer to the open file—the number passed to the **Open** statement.

The value returned depends on the mode in which the file was opened:

File Mode	Returns
Input	Byte position for the next read
Output	Byte position for the next write
Append	Byte position for the next write
Random	Number of the next record to be written or read
Binary	Byte position for the next read or write

The value returned is a **Long** between 1 and 2147483647, where the first byte (or first record) in the file is 1.

Example

```
'This example opens a file for random write, then writes ten
'records into the file using the Put statement. The file
'position is displayed using the Seek function, and the file is
'closed.
Sub Main()
   Open "test.dat" For Random Access Write As #1
   For x = 1 To 10
      r% = x * 10
      Put #1,x,r%
   Next x
   y = Seek(1)
   MsgBox "The current file position is: " & y
   Close
End Sub
```

See Also **Seek** (statement); **Loc** (function).

Platform(s) All.

Seek (statement)

Syntax `Seek [#]` *filenumber*`,` *position*

Description Sets the position of the file pointer within a given file such that the next read or write operation will occur at the specified position.

Comments The **Seek** statement accepts the following parameters:

Parameter	Description
filenumber	**Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.
position	**Long** that specifies the location within the file at which to position the file pointer. The value must be between 1 and 2147483647, where the first byte (or record number) in the file is 1. For files opened in either **Binary**, **Output**, **Input**, or **Append** mode, *position* is the byte position within the file. For **Random** files, *position* is the record number.

A file can be extended by seeking beyond the end of the file and writing data there.

Example

```
'This example opens a file for random write, then writes ten
'records into the file using the Put statement. The file is then
'reopened for read, and the ninth record is read using the Seek
'and Get functions.
Sub Main()
   Open "test.dat" For Random Access Write As #1
   For x = 1 To 10
      rec$ = "Record#: " & x
      Put #1,x,rec$
   Next x
   Close
   Open "test.dat" For Random Access Read As #1
   Seek #1,9
   Get #1,,rec$
   MsgBox "The ninth record = " & x
   Close
   Kill "test.dat"
End Sub
```

See Also **Seek** (function); **Loc** (function).

Platform(s) All.

Select...Case (statement)

Syntax

```
Select Case testexpression
[Case expressionlist
  [statement_block]]
[Case expressionlist
  [statement_block]]
   .
   .
```

```
[Case Else
  [statement_block]]
End Select
```

Description Used to execute a block of BasicScript statements depending on the value of a given expression.

Comments The **Select Case** statement has the following parts:

Part	Description
testexpression	Any numeric or string expression.
statement_block	Any group of BasicScript statements. If the *testexpression* matches any of the expressions contained in *expressionlist*, then this statement block will be executed.
expressionlist	A comma-separated list of expressions to be compared against *testexpression* using any of the following syntaxes: *expression* `[,expression]...` *expression* **To** *expression* **Is** *relational_operator expression* The resultant type of expression in *expressionlist* must be the same as that of *testexpression*.

Multiple expression ranges can be used within a single **Case** clause. For example:

```
Case 1 to 10,12,15, Is > 40
```

Only the *statement_block* associated with the first matching expression will be executed. If no matching *statement_block* is found, then the statements following the **Case Else** will be executed.

A **Select...End Select** expression can also be represented with the **If...Then** expression. The use of the **Select** statement, however, may be more readable.

Example

```
'This example uses the Select...Case statement to output the
'current operating system.
Sub Main()
   OpSystem% = Basic.OS
   Select Case OpSystem%
      Case 0,2
         s = "Microsoft Windows"
      Case 3 to 8, 12
         s = "UNIX"
      Case 10
         s = "IBM OS/2"
      Case Else
         s = "Other"
   End Select
   MsgBox "This version of BasicScript is running on: " & s
```

```
End Sub
```

See Also **Choose** (function); **Switch** (function); **IIf** (function); **If...Then...Else** (statement).

Platform(s) All.

SelectBox (function)

Syntax `SelectBox([`*title*`],`*prompt*`,`*ArrayOfItems* `[,`*helpfile*`,`*context*`])`

Description Displays a dialog box that allows the user to select from a list of choices and returns an **Integer** containing the index of the item that was selected.

Comments The **SelectBox** statement accepts the following parameters:

Parameter	Description
title	Title of the dialog box. This can be an expression convertible to a **String**. A runtime error is generated if *title* is **Null**. If **title** is missing, then the default title is used.
prompt	Text to appear immediately above the list box containing the items. This can be an expression convertible to a **String**. A runtime error is generated if *prompt* is **Null**.
ArrayOfItems	Single-dimensioned array. Each item from the array will occupy a single entry in the list box. A runtime error is generated if *ArrayOfItems* is not a single-dimensioned array. *ArrayOfItems* can specify an array of any fundamental data type (structures are not allowed). **Null** and **Empty** values are treated as zero-length strings.
helpfile	Name of the file containing context-sensitive help for this dialog. If this parameter is specified, then *context* must also be specified.
context	Number specifying the ID of the topic within *helpfile* for this dialog's help. If this parameter is specified, then *helpfile* must also be specified.

The value returned is an **Integer** representing the index of the item in the list box that was selected relative to the lower bound of *ArrayOfItems*. If the user selects Cancel, a value 1 less than the lower bound of the array is returned.

If both the *helpfile* and *context* parameters are specified, then a Help button is added in addition to the OK and Cancel buttons. Context-sensitive help can be invoked by selecting this button or using the help key (F1 on most platforms). Invoking help does not remove the dialog.

Example

```
'This example gets the current apps running, puts them in to an
'array and then asks the user to select one from a list.
Sub Main()
   Dim a$()
   AppList a$
   result% = SelectBox("Picker","Pick an application:",a$)
   If Not result% = -1 then
      Msgbox "User selected: " & a$(result%)
   Else
      Msgbox "User canceled"
   End If
End Sub
```

See Also **MsgBox** (statement); **AskBox, AskBox$** (functions); **AskPassword, AskPassword$** (functions); **InputBox**, **InputBox$** (functions); **OpenFileName$** (function); **SaveFileName$** (function); **AnswerBox** (function).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

SelectButton (statement)

Syntax `SelectButton` *name$* | *id*

Description Simulates a mouse click on the a push button given the push button's name (the *name$* parameter) or ID (the *id* parameter).

Comments The **SelectButton** statement accepts the following parameters:

Parameter	Description
name$	**String** containing the name of the push button to be selected.
id	**Integer** representing the ID of the push button to be selected.

A runtime error is generated if a push button with the given name or ID cannot be found in the active window.

Note: The **SelectButton** statement is used to select a button in another application's dialog box. This command is not intended for use with built-in or dynamic dialog boxes.

Example

```
'This example simulates the selection of several buttons in a
'dialog.
Sub Main()
   SelectButton "OK"
   SelectButton 2
   SelectButton "Close"
End Sub
```

See Also **ButtonEnabled** (function); **ButtonExists** (function).

Platform(s) Windows.

SelectComboBoxItem (statement)

Syntax `SelectComboBoxItem {`*name$* `|` *id*`},{`*ItemName$* `|` *ItemNumber*`} [,`*isDoubleClick*`]`

Description Selects an item from a combo box given the name or ID of the combo box and the name or line number of the item.

Comments The **SelectComboBoxItem** statement accepts the following parameters:

Parameter	Description
name$	**String** indicating the name of the combo box containing the item to be selected. The name of a combo box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a combo box. A runtime error is generated if a combo box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the combo box containing the item to be selected.
ItemName$	**String** specifying which item is to be selected. The string is compared without regard to case. If *ItemName$* is a zero-length string, then all currently selected items are deselected. A runtime error results if *ItemName$* cannot be found in the combo box.
ItemNumber	**Integer** containing the index of the item to be selected. A runtime error is generated if *ItemNumber* is not within the correct range.
isDoubleClick	**Boolean** value indicating whether a double click of that item is to be simulated.

Note: The **SelectComboBoxItem** statement is used to set the item of a combo box in another application's dialog box. Use the **DlgText** statement to change the content of the text box part of a list box in a dynamic dialog box.

Example

```
'This example simulates the selection of a couple of combo boxes.
Sub Main()
   SelectComboBoxItem "ComboBox1","Item4"
   SelectComboBoxItem 1,2,TRUE
```

```
End Sub
```

See Also **ComboBoxEnabled** (function); **ComboBoxExists** (function); **GetComboBoxItem$** (function); **GetComboBoxItemCount** (function).

Platform(s) Windows.

SelectListBoxItem (statement)

Syntax `SelectListBoxItem {`*name$* `|` *id*`},{`*ItemName$* `|` *ItemNumber*`} [,`*isDoubleClick*`]`

Description Selects an item from a list box given the name or ID of the list box and the name or line number of the item.

Comments The **SelectListBoxItem** statement accepts the following parameters:

Parameter	Description
name$	**String** indicating the name of the list box containing the item to be selected. The name of a list box is determined by scanning the window list looking for a text control with the given name that is immediately followed by a list box. A runtime error is generated if a list box with that name cannot be found within the active window.
id	**Integer** specifying the ID of the list box containing the item to be selected.
ItemName$	**String** specifying which item is to be selected. The string is compared without regard to case. If *ItemName$* is a zero-length string, then all currently selected items are deselected. A runtime error results if *ItemName$* cannot be found in the list box.
ItemNumber	**Integer** containing the index of the item to be selected. A runtime error is generated if *ItemNumber* is not within the correct range.
isDoubleClick	**Boolean** value indicating whether a double click of that item is to be simulated.

The list box must exist within the current window or dialog box; otherwise, a runtime error will be generated.

For multiselect list boxes, **SelectListBoxItem** will select additional items (i.e., it will not remove the selection from the currently selected items).

Note: The **SelectListBoxItem** statement is used to select an item in a list box of another application's dialog box. Use the **DlgText** statement to change the selected item in a list box within a dynamic dialog box.

Example

```
'This example simulates a double click on the first item in list
'box 1.
Sub Main()
   SelectListBoxItem "ListBox1",1,TRUE
End Sub
```

See Also **GetListBoxItem$** (function); **GetListBoxItemCount** (function); **ListBoxEnabled** (function); **ListBoxExists** (function).

Platform(s) Windows.

SendKeys (statement)

Syntax `SendKeys` *string* `[, [`*wait*`] [,`*delay*`]]`

Description Sends the specified keys to the active application, optionally waiting for the keys to be processed before continuing.

Comments The **SendKeys** statement accepts the following named parameters:

Named Parameter	Description
string	**String** containing the keys to be sent. The format for *string* is described below.
wait	**Boolean** value. If **True**, then BasicScript waits for the keys to be completely processed before continuing. The default value is **False**, which causes BasicScript to continue script execution while before **SendKeys** finishes.
delay	**Integer** specifying the number of milliseconds devoted for the output of the entire *string* parameter. It must be within the following range: `0 <=` *delay* `<= 32767` For example, if *delay* is 5000 (5 seconds) and the *string* parameter contains ten keys, then a key will be output every 1/2 second. If unspecified (or 0), the keys will play back at full speed.

The **SendKeys** statement will wait for a prior **SendKeys** to complete before executing.

Specifying Keys

To specify any key on the keyboard, simply use that key, such as "a" for lowercase a, or "A" for uppercase a.

Sequences of keys are specified by appending them together: "abc" or "dir /w".

Some keys have special meaning and are therefore specified in a special way—by enclosing them within braces. For example, to specify the percent sign, use "{%}". The following table shows the special keys:

Key	Special Meaning	Example	
+	Shift	"+{F1}"	Shift+F1
^	Ctrl	"^a"	Ctrl+A
~	Shortcut for Enter	"~"	Enter
%	Alt	"%F"	Alt+F
[]	No special meaning	"{[}"	Open bracket
{}	Used to enclose special keys	"{Up}"	Up arrow
()	Used to specify grouping	"^(ab)"	Ctrl+A, Ctrl+B

Keys that are not displayed when you press them are also specified within braces, such as {Enter} or {Up}. A list of these keys follows:

{BkSp}	{BS}	{Break}	{CapsLock}	{Clear}
{Delete}	{Del}	{Down}	{End}	{Enter}
{Escape}	{Esc}	{Help}	{Home}	{Insert}
{Left}	{NumLock}	{NumPad0}	{NumPad1}	{NumPad2}
{NumPad3}	{NumPad4}	{NumPad5}	{NumPad6}	{NumPad7}
{NumPad8}	{NumPad9}	{NumPad/}	{NumPad*}	{NumPad-}
{NumPad+}	{NumPad.}	{PgDn}	{PgUp}	{PrtSc}
{Right}	{Tab}	{Up}	{F1}	{Scroll Lock}
{F2}	{F3}	{F4}	{F5}	{F6}
{F7}	{F8}	{F9}	{F10}	{F11}
{F12}	{F13}	{F14}	{F15}	{F16}

Keys can be combined with Shift, Ctrl, and Alt using the reserved keys "+", "^", and "%" respectively:

For Key Combination	Use
Shift+Enter	"+{Enter}"
Ctrl+C	"^c"
Alt+F2	"%{F2}"

To specify a modifier key combined with a sequence of consecutive keys, group the key sequence within parentheses, as in the following example:

For Key Combination	Use
Shift+A, Shift+B	"+(abc)"
Ctrl+F1, Ctrl+F2	"^({F1}{F2})"

Use "~" as a shortcut for embedding Enter within a key sequence:

For Key Combination	Use
a, b, Enter, d, e	"ab~de"
Enter, Enter	"~~"

To embed quotation marks, use two quotation marks in a row:

For Key Combination	Use
"Hello"	""Hello""
a"b"c	"a""b""c"

Key sequences can be repeated using a repeat count within braces:

For Key Combination	Use
Ten "a" keys	"{a 10}"
Two Enter keys	"{Enter 2}"

Example

```
'This example runs Notepad, writes to Notepad, and saves the new
'file using the SendKeys statement.
Sub Main()
   id = Shell("Notepad.exe")
   AppActivate "Notepad"
   SendKeys "Hello, Notepad.",True  'Write some text.
   SendKeys "%fs",True              'Save File as "name.txt"
   SendKeys "name.txt{ENTER}",True
   AppClose "Notepad"
End Sub
```

See Also **DoKeys** (statement); **QueKeys** (statement); **QueKeyDn** (statement); **QueKeyUp** (statement).

Platform(s) Windows, Win32.

Set (statement)

Syntax 1 `Set` *object_var* `=` *object_expression*

Syntax 2 `Set` *object_var* `= New` *object_type*

Syntax 3 `Set` *object_var* `= Nothing`

Description Assigns a value to an object variable.

Comments

Syntax 1

The first syntax assigns the result of an expression to an object variable. This statement does not duplicate the object being assigned but rather copies a reference of an existing object to an object variable.

The *object_expression* is any expression that evaluates to an object of the same type as the *object_var*.

With data objects, **Set** performs additional processing. When the **Set** is performed, the object is notified that a reference to it is being made and destroyed. For example, the following statement deletes a reference to object A, then adds a new reference to B.

```
Set a = b
```

In this way, an object that is no longer being referenced can be destroyed.

Syntax 2

In the second syntax, the object variable is being assigned to a new instance of an existing object type. This syntax is valid only for data objects.

When an object created using the **New** keyword goes out of scope (i.e., the **Sub** or **Function** in which the variable is declared ends), the object is destroyed.

Syntax 3

The reserved keyword **Nothing** is used to make an object variable reference no object. At a later time, the object variable can be compared to **Nothing** to test whether the object variable has been instantiated:

```
Set a = Nothing
   :
If a Is Nothing Then Beep
```

Example

```
'This example creates two objects and sets their values.
Sub Main()
   Dim document As Object
   Dim page As Object
   Set document = GetObject("c:\resume.doc")
```

```
   Set page = Document.ActivePage
   MsgBox page.name
End Sub
```

See Also = (statement); **Let** (statement); **CreateObject** (function); **GetObject** (function).

Platform(s) All.

SetAttr (statement)

Syntax `SetAttr` *pathname*, *attributes*

Description Changes the attribute *pathname* to the given attribute. A runtime error results if the file cannot be found.

Comments The **SetAttr** statement accepts the following named parameters:

Named Parameter	Description
pathname	**String** containing the name of the file.
attributes	**Integer** specifying the new attribute of the file.

The *attributes* parameter can contain any combination of the following values:

Constant	Value	Includes
ebNormal	0	Turns off all attributes
ebReadOnly	1	Read-only files
ebHidden	2	Hidden files
ebSystem	4	System files
ebVolume	8	Volume label
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

The attributes can be combined using the + operator or the binary **Or** operator.

Example

```
'This example creates a file and sets its attributes to
'Read-Only and System.
Sub Main()
   Open "test.dat" For Output Access Write As #1
   Close
   MsgBox "The current file attribute is: " & GetAttr("test.dat")
   SetAttr "test.dat",ebReadOnly Or ebSystem
   MsgBox "The file attribute was set to: " & GetAttr("test.dat")
End Sub
```

See Also **GetAttr** (function); **FileAttr** (function).

Platform(s) All.

Platform Notes **Windows:** Under Windows, these attributes are the same as those used by DOS.

UNIX: On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

SetCheckBox (statement)

Syntax `SetCheckBox {`*name$* `|` *id*`},`*state*

Description Sets the state of the check box with the given name or ID.

Comments The **SetCheckBox** statement accepts the following parameters:

Parameter	Description
name$	**String** containing the name of the check box to be set.
id	**Integer** specifying the ID of the check box to be set.
state	**Integer** indicating the new state of the check box. If *state* is 1, then the box is checked. If *state* is 0, then the check is removed. If *state* is 2, then the box is dimmed (only applicable for three-state check boxes).

A runtime error is generated if a check box with the specified name cannot be found in the active window.

This statement has the side effect of setting the focus to the given check box.

Note: The **SetCheckBox** statement is used to set the state of a check box in another application's dialog box. Use the **DlgValue** statement to modify the state of a check box within a dynamic dialog box.

Example

```
'This example sets a check box.
Sub Main()
   SetCheckBox "CheckBox1",1
End Sub
```

See Also **CheckBoxExists** (function); **CheckBoxEnabled** (function); **GetCheckBox** (function); **DlgValue** (statement).

Platform(s) Windows.

SetEditText (statement)

Syntax `SetEditText {`*name$* `|` *id*`},`*content$*

Description Sets the content of an edit control given its name or ID.

Comments The **SetEditText** statement accepts the following parameters:

Parameter	Description
name$	**String** containing the name of the text box to be set. The name of a text box control is determined by scanning the window list looking for a text control with the given name that is immediately followed by an edit control. A runtime error is generated if a text box control with that name cannot be found within the active window.
id	**Integer** specifying the ID of the text box to be set. For text boxes that do not have a preceding text control, the *id* can be used to absolutely reference the control. The *id* is determined by examining the dialog box with a resource editor or using an application such as Spy.
content$	**String** containing the new content for the text box.

This statement has the side effect of setting the focus to the given text box.

Note: The **SetEditText** statement is used to set the content of a text box in another application's dialog box. Use the **DlgText** statement to set the text of a text box within a dynamic dialog box.

Example

```
'This example sets the content of the filename text box of the
'current window to "test.dat".
Sub Main()
   SetEditText "Filename:","test.dat"
End Sub
```

See Also **EditEnabled** (function); **EditExists** (function); **GetEditText$** (function).

Platform(s) Windows.

SetOption (statement)

Syntax `SetOption` *name$* | *id*

Description Selects the specified option button given its name or ID.

Comments The **SetOption** statement accepts the following parameters:

Parameter	Description
name$	**String** containing the name of the option button to be selected.

Parameter	Description
id	**Integer** containing the ID of the option button to be selected.

A runtime error is generated if the option button cannot be found within the active window.

Note: The **SetOption** statement is used to select an option button in another application's dialog box. Use the **DlgValue** statement to select an option button within a dynamic dialog box.

Example

```
'This example selects the Continue option button.
Sub Main()
   SetOption "Continue"
End Sub
```

See Also **GetOption** (function); **OptionEnabled** (function); **OptionExists** (function).

Platform(s) Windows.

Sgn (function)

Syntax `Sgn(`*number*`)`

Description Returns an **Integer** indicating whether a number is less than, greater than, or equal to 0.

Comments Returns 1 if *number* is greater than 0.

Returns 0 if *number* is equal to 0.

Returns –1 if *number* is less than 0.

The *number* parameter is a numeric expression of any type. If number is **Null**, then a runtime error is generated. **Empty** is treated as 0.

Example

```
'This example tests the product of two numbers and displays a
'message based on the sign of the result.
Sub Main()
   a% = -100
   b% = 100
   c% = a% * b%
   Select Case Sgn(c%)
      Case -1
         MsgBox "The product is negative " & Sgn(c%)
      Case 0
         MsgBox "The product is 0 " & Sgn(c%)
      Case 1
         MsgBox "The product is positive " & Sgn(c%)
   End Select
```

```
End Sub
```

See Also **Abs** (function).

Platform(s) All.

Shell (function)

Syntax `Shell(`*pathname* `[,`*windowstyle*`])`

Description Executes another application, returning the task ID if successful.

Comments The **Shell** statement accepts the following named parameters:

Named Parameter	Description	
pathname	**String** containing the name of the application and any parameters.	
windowstyle	Optional **Integer** specifying the state of the application window after execution. It can be any of the following values:	
	ebHide	Application is hidden.
	ebNormalFocus	Application is displayed in default position with the focus.
	ebMinimizedFocus	Application is minimized with the focus (this is the default).
	MaximizedFocus	Application is maximized with the focus.
	ebNormalNoFocus	Application is displayed in default position without the focus.
	ebMinimizedNoFocus	Application is minimized without the focus
	A runtime error is generated if *windowstyle* is not one of the above values.	

An error is generated if unsuccessful running *pathname*.

The **Shell** command runs programs asynchronously: the statement following the **Shell** statement will execute before the child application has exited. On some platforms, the next statement will run even before the child application has finished loading.

The **Shell** function returns a value suitable for activating the application using the **AppActivate** statement. It is important that this value be placed into a **Variant**, as its type depends on the platform.

Example

```
'This example displays the Windows Clock, delays a while, then
'closes it.
```

```
Sub Main()
   id = Shell("clock.exe",1)
   AppActivate "Clock"
   Sleep(2000)
   AppClose "Clock"
End Sub
```

See Also **PrintFile** (function); **SendKeys** (statement); **AppActivate** (statement).

Platform(s) All.

Platform Notes **Windows:** Under Windows, this function returns the hInstance of the application. Since this value is only a **WORD** in size, the upper **WORD** of the result is always zero.

The **Shell** function under Windows supports file associations. In other words, you can specify the name of a file, and the **Shell** function executes the associated application with that file as a parameter. (File associations are specified in the WIN.INI file.)

Win32: Under Win32, this function returns a global process ID that can be used to identify the new process. Under Win32, the **Shell** function does not support file associations (i.e., setting pathname to **"sample.txt"** will not execution Notepad).

When specifying long filenames as parameters, you may have to enclose the parameters in double quotes. For example, under Windows 95, to run WordPad, passing it a file called "Sample Document", you would use the following statement:

```
r = Shell("WordPad ""Sample Document""")
```

Macintosh: The Macintosh does not support wildcard characters such as * and ?. These are valid filename characters. Instead of wildcards, the Macintosh uses the **MacID** function to specify a collection of files of the same type. The syntax for this function is:

```
Shell(MacID(text$) [,windowstyle])
```

The *text$* parameter is a four-character string containing an application signature. A runtime error occurs if the **MacID** function is used on platforms other than the Macintosh.

On the Macintosh, the *windowstyle* parameter only specifies whether the application receives the focus.

UNIX: Under all versions of UNIX, the *windowstyle* parameter is ignored. This function returns the process identifier of the new process.

Under UNIX, BasicScript attempts to execute the command line using one of the installed shells. BasicScript looks for a shell using the following precedence:

1. BasicScript examines the SHELL environment variable, which is normally set to the path of the currently executing shell (e.g., **/bin/sh**, **/bin/csh**, and so on).
2. BasicScript examines the PATH environment variable for an executable program called **sh** (the Bourne shell).

3. In the unlikely event that a shell was not located with the above rules, BasicScript will search for **sh** in the following areas:
   ```
   /bin
   /usr/bin
   /usr/sbin
   ```

Once a suitable shell has been located, it is executed with *pathname* as a parameter. The environment of the calling process is made available to the new process and will be use by the shell in a manner specific to that shell.

Due to the asynchronous nature of the shell process, failure to find and start the program is not reported to BasicScript.

OS/2: Under OS/2, the **Shell** function is capable of running both Presentation Manager applications and command line applications. When running command line applications, the **Shell** function always returns 0.

Sin (function)

Syntax `Sin(`*number*`)`

Description Returns a **Double** value specifying the sine of *number*.

Comments The *number* parameter is a **Double** specifying an angle in radians.

Example
```
'This example displays the sine of pi/4 radians (45 degrees).
Sub Main()
   c# = Sin(Pi / 4)
   MsgBox "The sine of 45 degrees is: " & c#
End Sub
```

See Also **Tan** (function); **Cos** (function); **Atn** (function).

Platform(s) All.

Single (data type)

Syntax `Single`

Description A data type used to declare variables capable of holding real numbers with up to seven digits of precision.

Comments Single variables are used to hold numbers within the following ranges:

Sign	Range
Negative	-3.402823E38 <= *single* <= -1.401298E-45
Positive	1.401298E-45 <= *single* <= 3.402823E38

The type-declaration character for **Single** is **!**.

Storage

Internally, singles are stored as 4-byte (32-bit) IEEE values. Thus, when appearing within a structure, singles require 4 bytes of storage. When used with binary or random files, 4 bytes of storage is required.

Each single consists of the following

- A 1-bit sign
- An 8-bit exponent
- A 24-bit mantissa

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **String** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CSng** (function).

Platform(s) All.

Sleep (statement)

Syntax `Sleep` *milliseconds*

Description Causes the script to pause for a specified number of milliseconds.

Comments The *milliseconds* parameter is a **Long** in the following range:

`0 <=` *milliseconds* `<= 2,147,483,647`

Example

```
'This example displays a message for 2 seconds.
Sub Main()
   Msg.Open "Waiting 2 seconds",0,False,False
   Sleep(2000)
   Msg.Close
End Sub
```

Platform(s) All.

Platform Notes **Windows:** Under Windows, the accuracy of the system clock is modulo 55 milliseconds. The value of *milliseconds* will, in the worst case, be rounded up to the nearest multiple of 55. In other words, if *milliseconds* is 1, it will be rounded to 55 in the worst case.

Sln (function)

Syntax `Sln(`*cost*`, `*salvage*`, `*life*`)`

Description Returns the straight-line depreciation of an asset assuming constant benefit from the asset.

Comments The **Sln** of an asset is found by taking an estimate of its useful life in years, assigning values to each year, and adding up all the numbers.

The formula used to find the **Sln** of an asset is as follows:

```
(Cost - Salvage Value) / Useful Life
```

The **Sln** function requires the following named parameters:

Named Parameter	Description
cost	**Double** representing the initial cost of the asset.
salvage	**Double** representing the estimated value of the asset at the end of its useful life.
life	**Double** representing the length of the asset's useful life.

The unit of time used to express the useful life of the asset is the same as the unit of time used to express the period for which the depreciation is returned.

Example

```
'This example calculates the straight-line depreciation of an
'asset that cost $10,000.00 and has a salvage value of $500.00
'as scrap after ten years of service life.
Sub Main()
   dep# = Sln(10000.00,500.00,10)
   MsgBox "The annual depreciation is: " &
Format(dep#,"Currency")
End Sub
```

See Also **SYD** (function); **DDB** (function).

Platform(s) All.

Space, Space$ (functions)

Syntax `Space[$](`*number*`)`

Description Returns a string containing the specified number of spaces.

Comments **Space$** returns a **String**, whereas **Space** returns a **String** variant.

The *number* parameter is an **Integer** between 0 and 32767.

Example

```
'This example returns a string of ten spaces and displays it.
Sub Main()
   ln$ = Space$(10)
   MsgBox "Hello" & ln$ & "over there."
End Sub
```

See Also **String**, **String$** (functions); **Spc** (function).

Platform(s) All.

Spc (function)

Syntax `Spc(`*numspaces*`)`

Description Prints out the specified number of spaces. This function can only be used with the **Print** and **Print#** statements.

Comments The *numspaces* parameter is an **Integer** specifying the number of spaces to be printed. It can be any value between 0 and 32767.

If a line width has been specified (using the **Width** statement), then the number of spaces is adjusted as follows:

```
numspaces = numspaces Mod width
```

If the resultant number of spaces is greater than width – print_position, then the number of spaces is recalculated as follows:

```
numspaces = numspaces – (width – print_position)
```

These calculations have the effect of never allowing the spaces to overflow the line length. Furthermore, with a large value for column and a small line width, the file pointer will never advance more than one line.

Example

```
'This example displays 20 spaces between the arrows.
Sub Main()
   Viewport.Open
   Print "I am"; Spc(20); "20 spaces apart!"
   Sleep (10000)'Wait 10 seconds.
   Viewport.Close
End Sub
```

See Also **Tab** (function); **Print** (statement); **Print#** (statement).

Platform(s) All.

SQLBind (function)

Syntax `SQLBind(`*connectionnum, array* `[,`*column*`])`

Description Specifies which fields are returned when results are requested using the **SQLRetrieve** or **SQLRetrieveToFile** function.

Comments The following table describes the named parameters to the **SQLBind** function:

Named Parameter	Description
connectionnum	**Long** parameter specifying a valid connection.
array	Any array of variants. Each call to **SQLBind** adds a new column number (an **Integer**) in the appropriate slot in the array. Thus, as you bind additional columns, the *array* parameter grows, accumulating a sorted list (in ascending order) of bound columns. If *array* is fixed, then it must be a one-dimensional variant array with sufficient space to hold all the bound column numbers. A runtime error is generated if *array* is too small. If *array* is dynamic, then it will be resized to exactly hold all the bound column numbers.
column	Optional **Long** parameter that specifies the column to which to bind data. If this parameter is omitted, all bindings for the connection are dropped.

This function returns the number of bound columns on the connection. If no columns are bound, then 0 is returned. If there are no pending queries, then calling **SQLBind** will cause an error (queries are initiated using the **SQLExecQuery** function).

If supported by the driver, row numbers can be returned by binding column 0.

BasicScript generates a trappable runtime error if **SQLBind** fails. Additional error information can then be retrieved using the **SQLError** function.

Example

```
'This example binds columns to data.
Sub Main()
   Dim columns() As Variant
   id& = SQLOpen("dsn=SAMPLE",,3)
   t& = SQLExecQuery(id&,"Select * From c:\sample.dbf")
   i% = SQLBind(id&,columns,3)
   i% = SQLBind(id&,columns,1)
   i% = SQLBind(id&,columns,2)
   i% = SQLBind(id&,columns,6)
   For x = 0 To (i% - 1)
      MsgBox columns(x)
   Next x
   id& = SQLClose(id&)
End Sub
```

See Also **SQLRetrieve** (function); **SQLRetrieveToFile** (function).

Platform(s) Windows, Win32.

SQLClose (function)

Syntax `SQLClose(`*connectionnum*`)`

Description Closes the connection to the specified data source.

Comments The unique connection ID (*connectionnum*) is a **Long** value representing a valid connection as returned by **SQLOpen**. After **SQLClose** is called, any subsequent calls made with the *connectionnum* will generate runtime errors.

The **SQLClose** function returns 0 if successful; otherwise, it returns the passed connection ID and generates a trappable runtime error. Additional error information can then be retrieved using the **SQLError** function.

BasicScript automatically closes all open SQL connections when either the script or the application terminates. You should use the **SQLClose** function rather than relying on BasicScript to automatically close connections in order to ensure that your connections are closed at the proper time.

Example

```
'This example disconnects the the data source sample.
Sub Main()
   id& = SQLOpen("dsn=SAMPLE",,3)
   id& = SQLClose(id&)
End Sub
```

See Also **SQLOpen** (function).

Platform(s) Windows, Win32.

SQLError (function)

Syntax `SQLError(`*resultarray*`, `*connectionnum*`)`

Description Retrieves driver-specific error information for the most recent SQL functions that failed.

Comments

This function is called after any other SQL function fails. Error information is returned in a two-dimensional array (*resultarray*). The following table describes the named parameters to the **SQLError** function:

Named Parameter	Description
resultarray	Two-dimensional **Variant** array, which can be dynamic or fixed. If the array is fixed, it must be (*x*,3), where *x* is the number of errors you want returned. If *x* is too small to hold all the errors, then the extra error information is discarded. If *x* is greater than the number of errors available, all errors are returned, and the empty array elements are set to **Empty**. If the array is dynamic, it will be resized to hold the exact number of errors.
connectionnum	Optional **Long** parameter specifying a connection ID. If this parameter is omitted, error information is returned for the most recent SQL function call.

Each array entry in the *resultarray* parameter describes one error. The three elements in each array entry contain the following information:

Element	Value
(entry,0)	The ODBC error state, indicated by a **Long** containing the error class and subclass.
(entry,1)	The ODBC native error code, indicated by a **Long**.
(entry,2)	The text error message returned by the driver. This field is **String** type.

For example, to retrieve the ODBC text error message of the first returned error, the array is referenced as:

resultarray`(0,2)`

The **SQLError** function returns the number of errors found.

BasicScript generates a runtime error if **SQLError** fails. (You cannot use the **SQLError** function to gather additional error information in this case.)

Example

```
'This example forces a connection error and traps it for use
'with the SQLError function.
Sub Main()
   Dim a() As Variant
   On Error Goto Trap
   id& = SQLOpen("",,4)
   id& = SQLClose(id&)
   Exit Sub
Trap:
   rc% = SQLError(a)
```

```
   If (rc%) Then
      For x = 0 To (rc% - 1)
         MsgBox "The SQLState returned was: " & a(x,0)
         MsgBox "The native error code returned was: " & a(x,1)
         MsgBox a(x,2)
      Next x
   End If
End Sub
```

Platform(s) Windows, Win32.

SQLExecQuery (function)

Syntax `SQLExecQuery(`*connectionnum*`, `*querytext*`)`

Description Executes an SQL statement query on a data source.

Comments This function is called after a connection to a data source is established using the **SQLOpen** function. The **SQLExecQuery** function may be called multiple times with the same connection ID, each time replacing all results.

The following table describes the named parameters to the **SQLExecQuery** function:

Named Parameter	Description
connectionnum	**Long** identifying a valid connected data source. This parameter is returned by the **SQLOpen** function.
querytext	**String** specifying an SQL query statement. The SQL syntax of the string must strictly follow that of the driver.

The return value of this function depends on the result returned by the SQL statement:

SQL Statement	Value
SELECT...FROM	The value returned is the number of columns returned by the SQL statement
DELETE,INSERT,UPDATE	The value returned is the number of rows affected by the SQL statement

BasicScript generates a runtime error if **SQLExecQuery** fails. Additional error information can then be retrieved using the **SQLError** function.

Example

```
'This example executes a query on the connected data source.
Sub Main()
   Dim s As String
   Dim qry As Long
   Dim a() As Variant
   On Error Goto Trap
   id& = SQLOpen("dsn=SAMPLE", s$, 3)
```

```
   qry = SQLExecQuery(id&,"Select * From c:\sample.dbf")
   MsgBox "There are " & qry & " columns in the result set."
   id& = SQLClose(id&)
   Exit Sub
Trap:
   rc% = SQLError(a)
   If (rc%) Then
      For x = 0 To (rc% - 1)
         MsgBox "The SQLState returned was: " & a(x,0)
         MsgBox "The native error code returned was: " & a(x,1)
         MsgBox a(x,2)
      Next x
   End If
End Sub
```

See Also **SQLOpen** (function); **SQLClose** (function); **SQLRetrieve** (function); **SQLRetrieveToFile** (function).

Platform(s) Windows, Win32.

SQLGetSchema (function)

Syntax `SQLGetSchema(`*connectionnum*`, `*typenum*`, [, [`*resultarray*`] [, `*qualifiertext*`]])`

Description Returns information about the data source associated with the specified connection.

Comments The following table describes the named parameters to the **SQLGetSchema** function:

Named Parameter	Description
connectionnum	**Long** parameter identifying a valid connected data source. This parameter is returned by the **SQLOpen** function.

Named Parameter	Description
typenum	**Integer** parameter specifying the results to be returned. The following are the values for this parameter:

Value	Meaning
1	Returns a one-dimensional array of available data sources. The array is returned in the *resultarray* parameter.
2	Returns a one-dimensional array of databases (either directory names or database names, depending on the driver) associated with the current connection. The array is returned in the *resultarray* parameter.
3	Returns a one-dimensional array of owners (user IDs) of the database associated with the current connection. The array is returned in the *resultarray* parameter.
4	Returns a one-dimensional array of table names for a specified owner and database associated with the current connection. The array is returned in the *resultarray* parameter.
5	Returns a two-dimensional array (*n* by 2) containing information about a specified table. The first element contains the column name. The second element contains the data type of the column
6	Returns a string containing the ID of the current user.
7	Returns a string containing the name (either the directory name or the database name, depending on the driver) of the current database.

Named Parameter	Description	
	8	Returns a string containing the name of the data source on the current connection.
	9	Returns a string containing the name of the DBMS of the data source on the current connection (e.g., "FoxPro 2.5" or "Excel Files").
	10	Returns a string containing the name of the server for the data source.
	11	Returns a string containing the owner qualifier used by the data source (e.g., "owner," "Authorization ID," "Schema").
	12	Returns a string containing the table qualifier used by the data source (e.g., "table," "file").
	13	Returns a string containing the database qualifier used by the data source (e.g., "database," "directory").
	14	Returns a string containing the procedure qualifier used by the data source (e.g., "database procedure," "stored procedure," "procedure").
resultarray	Optional **Variant** array parameter. This parameter is only required for action values 1, 2, 3, 4, and 5. The returned information is put into this array. If *resultarray* is fixed and it is not the correct size necessary to hold the requested information, then **SQLGetSchema** will fail. If the array is larger than required, then any additional elements are erased. If *resultarray* is dynamic, then it will be redimensioned to hold the exact number of elements requested.	

Named Parameter	Description
qualifiertext	Optional **String** parameter required for actions 3, 4, or 5. The values are as follows:

Action	Qualifier
3	The *qualifiertext* parameter must be the name of the database represented by ID.
4	The *qualifiertext* parameter specifies a database name and an owner name. The syntax for this string is: *DatabaseName.OwnerName*
5	The *qualifiertext* parameter specifies the name of a table on the current connection.

BasicScript generates a runtime error if **SQLGetSchema** fails. Additional error information can then be retrieved using the **SQLError** function.

If you want to retrieve the available data sources (where *typenum* = 1) before establishing a connection, you can pass 0 as the *connectionnum* parameter. This is the only action that will execute successfully without a valid connection.

This function calls the ODBC functions **SQLGetInfo** and **SQLTables** in order to retrieve the requested information. Some database drivers do not support these calls and will therefore cause the **SQLGetSchema** function to fail.

Example

```
'This example gets all available data sources.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   Dim dsn() As Variant
   numdims% = SQLGetSchema(0,1,dsn)
   If (numdims%) Then
      message = "Valid data sources are:" & crlf
      For x = 0 To numdims% - 1
         message = message & dsn(x) & crlf
      Next x
   Else
      message = "There are no available data sources."
   End If
   MsgBox message
End Sub
```

See Also **SQLOpen** (function).

Platform(s) Windows, Win32.

SQLOpen (function)

Syntax `SQLOpen(`*connectionstr* `[, [`*outputref*`] [, `*driverprompt*`]])`

Description Establishes a connection to the specified data source, returning a **Long** representing the unique connection ID.

Comments This function connects to a data source using a login string (*connectionstr*) and optionally sets the completed login string (*outputref*) that was used by the driver. The following table describes the named parameters to the **SQLOpen** function:

Named Parameter	Description
connectionstr	**String** expression containing information required by the driver to connect to the requested data source. The syntax must strictly follow the driver's SQL syntax.
outputref	Optional **String** variable that will receive a completed connection string returned by the driver. If this parameter is missing, then no connection string will be returned.
driverprompt	**Integer** expression specifying any of the following values:

Value	Meaning
1	The driver's login dialog box is always displayed.
2	The driver's dialog box is only displayed if the connection string does not contain enough information to make the connection. This is the default behavior.
3	The driver's dialog box is only displayed if the connection string does not contain enough information to make the connection. Dialog box options that were passed as valid parameters are dimmed and unavailable.
4	The driver's login dialog box is never displayed.

The **SQLOpen** function will never return an invalid connection ID. The following example establishes a connection using the driver's login dialog box:

```
id& = SQLOpen("",,1)
```

BasicScript returns 0 and generates a trappable runtime error if **SQLOpen** fails. Additional error information can then be retrieved using the **SQLError** function.

Before you can use any SQL statements, you must set up a data source and relate an existing database to it. This is accomplished using the odbcadm.exe program.

Example

```
'This example connects the data source called "sample,"
'returning the completed connction string, and then displays it.
Sub Main()
   Dim s As String
   id& = SQLOpen("dsn=SAMPLE",s$,3)
   MsgBox "The completed connection string is: " & s$
   id& = SQLClose(id&)
End Sub
```

See Also **SQLClose** (function).

Platform(s) Windows, Win32.

SQLRequest (function)

Syntax `SQLRequest(`*connectionstr*`, `*querytext*`, `*resultarray*` [, [`*outputref*`] [, [`*driverprompt*`] [, `*colnameslogical*`]]])`

Description Opens a connection, runs a query, and returns the results as an array.

Comments The **SQLRequest** function takes the following named parameters:

Named Parameter	Description
connectionstr	**String** specifying the connection information required to connect to the data source.
querytext	**String** specifying the query to execute. The syntax of this string must strictly follow the syntax of the ODBC driver.
resultarray	Array of variants to be filled with the results of the query. The *resultarray* parameter must be dynamic: it will be resized to hold the exact number of records and fields.
outputref	Optional **String** to receive the completed connection string as returned by the driver.

Named Parameter	Description
driverprompt	Optional **Integer** specifying the behavior of the driver's dialog box:

Value	Meaning
1	The driver's login dialog box is always displayed.
2	The driver's dialog box is only displayed if the connection string does not contain enough information to make the connection. This is the default behavior.
3	The driver's dialog box is only displayed if the connection string does not contain enough information to make the connection. Dialog box options that were passed as valid parameters are dimmed and unavailable.
4	The driver's login dialog box is never displayed.

colnameslogical	Optional **Boolean** specifying whether the column names are returned as the first row of results. The default is **False**.

BasicScript generates a runtime error if **SQLRequest** fails. Additional error information can then be retrieved using the **SQLError** function.

The **SQLRequest** function performs one of the following actions, depending on the type of query being performed:

Type of Query	Action
SELECT	The **SQLRequest** function fills *resultarray* with the results of the query, returning a **Long** containing the number of results placed in the array. The array is filled as follows (assuming an *x* by *y* query):

Type of Query	Action
	(record 1,field 1) (record 1,field 2) : (record 1,field *y*) (record 2,field 1) (record 2,field 2) : (record 2,field *y*) : : (record *x*,field 1) (record *x*,field 2) : (record *x*,field *y*)
INSERT, DELETE, UPDATE	The **SQLRequest** function erases *resultarray* and returns a **Long** containing the number of affected rows.

Example

```
'This example opens a data source, runs a select query on it,
'and then displays all the data found in the result set.
Sub Main()
   Dim a() As Variant
   l& = SQLRequest("dsn=SAMPLE;","Select * From c:\sample.dbf" _
      ,a,,3,True)
   For x = 0 To Ubound(a)
      For y = 0 To l - 1
         MsgBox a(x,y)
      Next y
   Next x
End Sub
```

Platform(s) Windows, Win32.

SQLRetrieve (function)

Syntax `SQLRetrieve(`*connectionnum*`, `*resultarray*`[, [`*maxcolumns*`] [, [ `*maxrows*`] [, [`*colnameslogical*`] [, `*fetchfirstlogical*`]]]])`

Description Retrieves the results of a query.

Comments

This function is called after a connection to a data source is established, a query is executed, and the desired columns are bound. The following table describes the named parameters to the **SQLRetrieve** function:

Named Parameter	Description
connectionnum	**Long** identifying a valid connected data source with pending query results.
resultarray	Two-dimensional array of variants to receive the results. The array has *x* rows by *y* columns. The number of columns is determined by the number of bindings on the connection.
maxcolumns	Optional **Integer** expression specifying the maximum number of columns to be returned. If *maxcolumns* is greater than the number of columns bound, the additional columns are set to empty. If *maxcolumns* is less than the number of bound results, the rightmost result columns are discarded until the result fits.
maxrows	Optional **Integer** specifying the maximum number of rows to be returned. If *maxrows* is greater than the number of rows available, all results are returned, and additional rows are set to empty. If *maxrows* is less than the number of rows available, the array is filled, and additional results are placed in memory for subsequent calls to **SQLRetrieve**.
colnameslogical	Optional **Boolean** specifying whether column names should be returned as the first row of results. The default is **False**.
fetchfirstlogical	Optional **Boolean** expression specifying whether results are retrieved from the beginning of the result set. The default is **False**.

Before you can retrieve the results from a query, you must (1) initiate a query by calling the **SQLExecQuery** function and (2) specify the fields to retrieve by calling the **SQLBind** function.

This function returns a **Long** specifying the number of rows available in the array.

BasicScript generates a runtime error if **SQLRetrieve** fails. Additional error information is placed in memory.

Example

```
'This example executes a query on the connected data source,
'binds columns, and retrieves them.
Sub Main()
   Dim a() As Variant
   Dim b() As Variant
   Dim c() As Variant
   On Error Goto Trap
   id& = SQLOpen("DSN=SAMPLE",,3)
   qry& = SQLExecQuery(id&,"Select * From c:\sample.dbf"")
```

```
   i% = SQLBind(id&,b,3)
   i% = SQLBind(id&,b,1)
   i% = SQLBind(id&,b,2)
   i% = SQLBind(id&,b,6)
   l& = SQLRetrieve(id&,c)
   For x = 0 To Ubound(c)
      For y = 0 To l& - 1
         MsgBox c(x,y)
      Next y
   Next x
   id& = SQLClose(id&)
   Exit Sub
Trap:
   rc% = SQLError(a)
   If (rc%) Then
      For x = 0 To (rc% - 1)
         MsgBox "The SQLState returned was: " & a(x,0)
         MsgBox "The native error code returned was: " & a(x,1)
         MsgBox a(x,2)
      Next x
   End If
End Sub
```

See Also **SQLOpen** (function); **SQLExecQuery** (function); **SQLClose** (function); **SQLBind** (function); **SQLRetrieveToFile** (function).

Platform(s) Windows, Win32.

SQLRetrieveToFile (function)

Syntax `SQLRetrieveToFile(`*connectionnum*`, `*destination*` [, [`*colnameslogical*`] [, `*columndelimiter*`]])`

Description Retrieves the results of a query and writes them to the specified file.

Comments The following table describes the named parameters to the **SQLRetrieveToFile** function:

Named Parameter	Description
connectionnum	**Long** specifying a valid connection ID.
destination	**String** specifying the file where the results are written.
colnameslogical	Optional **Boolean** specifying whether the first row of results returned are the bound column names. By default, the column names are not returned.
columndelimiter	Optional **String** specifying the column separator. A tab (**Chr$(9)**) is used as the default.

Before you can retrieve the results from a query, you must (1) initiate a query by calling the **SQLExecQuery** function and (2) specify the fields to retrieve by calling the **SQLBind** function.

This function returns the number of rows written to the file. A runtime error is generated if there are no pending results or if BasicScript is unable to open the specified file.

BasicScript generates a runtime error if **SQLRetrieveToFile** fails. Additional error information may be placed in memory for later use with the **SQLError** function.

Example

```
'This example opens a connection, runs a query, binds columns,
'and writes the results to a file.
Sub Main()
   Dim a() As Variant
   Dim b() As Variant
   On Error Goto Trap
   id& = SQLOpen("DSN=SAMPLE;UID=RICH",,4)
   t& = SQLExecQuery(id&, "Select * From c:\sample.dbf"")
   i% = SQLBind(id&,b,3)
   i% = SQLBind(id&,b,1)
   i% = SQLBind(id&,b,2)
   i% = SQLBind(id&,b,6)
   l& = SQLRetrieveToFile(id&,"c:\results.txt",True,",")
   id& = SQLClose(id&)
   Exit Sub
Trap:
   rc% = SQLError(a)
   If (rc%) Then
      For x = 0 To (rc-1)
         MsgBox "The SQLState returned was: " & a(x,0)
         MsgBox "The native error code returned was: " & a(x,1)
         MsgBox a(x,2)
      Next x
   End If
End Sub
```

See Also **SQLOpen** (function); **SQLExecQuery** (function); **SQLClose** (function); **SQLBind** (function); **SQLRetrieve** (function).

Platform(s) Windows, Win32.

Sqr (function)

Syntax `Sqr(`*number*`)`

Description Returns a **Double** representing the square root of *number*.

Comments The *number* parameter is a **Double** greater than or equal to 0.

Example

```
'This example calculates the square root of the numbers from 1
'to 10 and displays them.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   For x = 1 To 10
      sx# = Sqr(x)
      message = message & Format(x,"Fixed") & " - " _
         & Format(sx#,"Fixed") & crlf
   Next x
   MsgBox message
End Sub
```

Platform(s) All.

Stop (statement)

Syntax `Stop`

Description Suspends execution of the current script, returning control to a debugger if one is present. If a debugger is not present, this command will have the same effect as **End**.

Example

```
'The Stop statement can be used for debugging. In this example,
'it is used to stop execution when Z is randomly set to 0.
Sub Main()
   For x = 1 To 10
      z = Random(0,10)
      If z = 0 Then Stop
      y = x / z
   Next x
End Sub
```

See Also **Exit For** (statement); **Exit Do** (statement); **Exit Function** (statement); **Exit Sub** (statement); **End** (statement).

Platform(s) All.

Str, Str$ (functions)

Syntax `Str[$](`*number*`)`

Description Returns a string representation of the given number.

Comments The *number* parameter is any numeric expression or expression convertible to a number. If *number* is negative, then the returned string will contain a leading minus sign. If *number* is positive, then the returned string will contain a leading space.

Singles are printed using only 7 significant digits. Doubles are printed using 15–16 significant digits.

These functions only output the period as the decimal separator and do not output thousands separators. Use the **CStr**, **Format**, or **Format$** function for this purpose.

Example

```
'In this example, the Str$ function is used to display the
'value of a numeric variable.
Sub Main()
   x# = 100.22
   MsgBox "The string value is: " + Str(x#)
End Sub
```

See Also **Format**, **Format$** (functions); **CStr** (function).

Platform(s) All.

StrComp (function)

Syntax `StrComp(`*string1*`,`*string2* `[,`*compare*`])`

Description Returns an **Integer** indicating the result of comparing the two string arguments.

Comments One of the following values is returned:

0	*string1* = *string2*
1	*string1* > *string2*
–1	*string1* < *string2*
Null	*string1* or *string2* is **Null**

The **StrComp** function accepts the following parameters:

Parameter	Description
string1	First string to be compared, which can be any expression convertible to a **String**.
string2	Second string to be compared, which can be any expression convertible to a **String**.
compare	Optional **Integer** specifying how the comparison is to be performed. It can be either of the following values: **0** Case-sensitive comparison **1** Case-insensitive comparison

Parameter	Description
	If *compare* is not specified, then the current **Option Compare** setting is used. If no **Option Compare** statement has been encountered, then **Binary** is used (i.e., string comparison is case-sensitive).

Example

```
'This example compares two strings and displays the results. It
'illustrates that the function compares two strings to the
'length of the shorter string in determining equivalency.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This string is UPPERCASE and lowercase"
   b$ = "This string is uppercase and lowercase"
   c$ = "This string"
   d$ = "This string is uppercase and lowercase characters"
   abc = StrComp(a$,b$,0)
   message = message & "a and c (sensitive)  : " & _
      Format(abc,"True/False") & crlf
   abi = StrComp(a$,b$,1)
   message = message & "a and b (insensitive): " & _
      Format(abi,"True/False") & crlf
   aci = StrComp(a$,c$,1)
   message = message & "a and c (insensitive): " & _
      Format(aci,"True/False") & crlf
   bdi = StrComp(b$,d$,1)
   message = message & "b and d (sensitive)  : " & _
      Format(bdi,"True/False") & crlf
   MsgBox message
End Sub
```

See Also Comparison Operators (topic); **Like** (operator); **Option Compare** (statement).

Platform(s) All.

StrConv (function)

Syntax `StrConv(`*string*`, `*conversion*`)`

Description Converts a string based on a conversion parameter.

Comments The StrConv function takes the following named parameters:

Named Parameter	Description
string	A **String** expression specifying the string to be converted.
conversion	An Integer specifying the types of conversions to be performed.

The *conversion* parameter can be any combination of the following constants:

Constant	Value	Description
ebUpperCase	1	Converts *string* to uppercase. This constant is supported on all platforms.
ebLowerCase	2	Converts *string* to lowercase. This constant is supported on all platforms.
ebProperCase	3	Capitalizes the first letter of each word and lower-cases all the letters. This constant is supported on all platforms.
ebWide	4	Converts narrow characters to wide characters. This constant is supported on Japanese locales only.
ebNarrow	8	Converts wide characters to narrow characters. This constant is supported on Japanese locales only.
ebKatakana	16	Converts Hiragana characters to Katakana characters. This constant is supported on Japanese locales only.
ebHiragana	32	Converts Katakana characters to Hiragana characters. This constant is supported on Japanese locales only.
ebUnicode	64	Converts string from MBCS to UNICODE. (This constant can only be used on platforms supporting UNICODE.)
ebFromUnicode	128	Converts string from UNICODE to MBCS. (This constant can only be used on platforms supporting UNICODE.)

A runtime error is generated when a conversion is requested that is not supported on the current platform. For example, the **ebWide** and **ebNarrow** constants can only be used on an MBCS platform. (You can determine platform capabilities using the **Basic.Capabilities** method.)

The following groupings of constants are mutually exclusive and therefore cannot be specified at the same time:

```
ebUpperCase, ebLowerCase, ebProperCase
ebWide, ebNarrow
ebUnicode, ebFromUnicode
```

Many of the constants can be combined. For example, **ebLowerCase Or ebNarrow**.

When converting to proper case (i.e., the **ebProperCase** constant), the following are seen as word delimiters: tab, linefeed, carriage-return, formfeed, vertical tab, space, null.

Example

```
Sub Main()
   a = InputBox("Type any string:")
   MsgBox "Upper case: " & StrConv(a,ebUpperCase)
   MsgBox "Lower case: " & StrConv(a,ebLowerCase)
   MsgBox "Proper case: " & StrConv(a,ebProperCase)
   If Basic.Capability(10) And Basic.OS = ebWin16 Then
      'This is an MBCS locale
      MsgBox "Narrow: " & StrConv(a,ebNarrow)
      MsgBox "Wide: " & StrConv(a,ebWide)
      MsgBox "Katakana: " & StrConv(a,ebKatakana)
      MsgBox "Hiragana: " & StrConv(a,ebHiragana)
   End If
End Sub
```

See Also **UCase, UCase$** (functions); **LCase, LCase$** (functions); **Basic.Capability** (method).

Platform(s) All.

String (data type)

Syntax

```
String
```

Description A data type capable of holding a number of characters.

Comments Strings are used to hold sequences of characters, each character having a value between 0 and 255. Strings can be any length up to a maximum length of 32767 characters.

Strings can contain embedded nulls, as shown in the following example:

```
s$ = "Hello" + Chr$(0) + "there"    'String with embedded
                                    'null
```

The length of a string can be determined using the **Len** function. This function returns the number of characters that have been stored in the string, including unprintable characters.

The type-declaration character for **String** is **$**.

String variables that have not yet been assigned are set to zero-length by default.

Strings are normally declared as variable-length, meaning that the memory required for storage of the string depends on the size of its content. The following BasicScript statements declare a variable-length string and assign it a value of length 5:

```
Dim s As String
s = "Hello" 'String has length 5.
```

Fixed-length strings are given a length in their declaration:

```
Dim s As String * 20
s = "Hello"              'String length = 20 with spaces to
                         'end of string.
```

When a string expression is assigned to a fixed-length string, the following rules apply:

- If the string expression is less than the length of the fixed-length string, then the fixed-length string is padded with spaces up to its declared length.
- If the string expression is greater than the length of the fixed-length string, then the string expression is truncated to the length of the fixed-length string.

Fixed-length strings are useful within structures when a fixed size is required, such as when passing structures to external routines.

The storage for a fixed-length string depends on where the string is declared, as described in the following table:

Strings Declared	Are Stored
In structures	In the same data area as that of the structure. Local structures are on the stack; public structures are stored in the public data space; and private structures are stored in the private data space. Local structures should be used sparingly as stack space is limited.
In arrays	In the global string space along with all the other array elements.
In local routines	On the stack. The stack is limited in size, so local fixed-length strings should be used sparingly.

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **Variant** (data type); **Boolean** (data type); **Def***Type* (statement); **CStr** (function).

Platform(s) All.

String, String$ (functions)

Syntax `String[$](`*number, character*`)`

Description Returns a string of length *number* consisting of a repetition of the specified filler character.

Comments **String$** returns a **String**, whereas **String** returns a **String** variant.

These functions take the following named parameters:

Named Parameter	Description
number	**Long** specifying the number of repetitions.
character	**Integer** specifying the character code to be used as the filler character. If *character* is greater than 255 (the largest character value), then BasicScript converts it to a valid character using the following formula: *character* `Mod 256` If character is a string, then the first character of that string is used as the filler character.

Example

```
'This example uses the String function to create a line of "="
'signs the length of another string and then displays the
'character string underlined with the generated string.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This string will appear underlined."
   b$ = String$(Len(a$),"=")
   MsgBox a$ & crlf & b$
End Sub
```

See Also **Space**, **Space$** (functions).

Platform(s) All.

Sub...End Sub (statement)

Syntax

```
[Private | Public] [Static] Sub name[(arglist)]
   [statements]
End Sub
```

where *arglist* is a comma-separated list of the following (up to 30 arguments are allowed):

```
[Optional] [ByVal | ByRef] parameter[()] [As type]
```

Description Declares a subroutine.

Comments The **Sub** statement has the following parts:

Part	Description
Private	Indicates that the subroutine being defined cannot be called from other scripts.
Public	Indicates that the subroutine being defined can be called from other scripts. If the **Private** and **Public** keywords are both missing, then **Public** is assumed.

Part	Description
Static	Recognized by the compiler but currently has no effect.
name	Name of the subroutine, which must follow BasicScript naming conventions: 1. Must start with a letter. 2. May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character. 3. Must not exceed 80 characters in length.
Optional	Keyword indicating that the parameter is optional. All optional parameters must be of type **Variant**. Furthermore, all parameters that follow the first optional parameter must also be optional. If this keyword is omitted, then the parameter is required. **Note:** You can use the **IsMissing** function to determine whether an optional parameter was actually passed by the caller.
ByVal	Keyword indicating that the parameter is passed by value.
ByRef	Keyword indicating that the parameter is passed by reference. If neither the **ByVal** nor the **ByRef** keyword is given, then **ByRef** is assumed.
parameter	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of **As** *type*.

type	Type of the parameter (i.e., **Integer**, **String**, and so on). Arrays are indicated with parentheses. For example, an array of integers would be declared as follows
	`Sub Test(a() As Integer)`
	`End Sub`

A subroutine terminates when one of the following statements is encountered:

```
End Sub
Exit Sub
```

Subroutines can be recursive.

Passing Parameters to Subroutines

Parameters are passed to a subroutine either by value or by reference, depending on the declaration of that parameter in *arglist*. If the parameter is declared using the **ByRef** keyword, then any modifications to that passed parameter within the subroutine change

the value of that variable in the caller. If the parameter is declared using the **ByVal** keyword, then the value of that variable cannot be changed in the called subroutine. If neither the **ByRef** nor the **ByVal** keyword is specified, then the parameter is passed by reference.

You can override passing a parameter by reference by enclosing that parameter within parentheses. For instance, the following example passes the variable j by reference, regardless of how the third parameter is declared in the *arglist* of **UserSub**:

```
UserSub 10,12,(j)
```

Optional Parameters

BasicScript allows you to skip parameters when calling subroutines, as shown in the following example:

```
Sub Test(a%,b%,c%)
End Sub
Sub Main
   Test 1,,4      'Parameter 2 was skipped.
End Sub
```

You can skip any parameter with the following restrictions:

1. The call cannot end with a comma. For instance, using the above example, the following is not valid:

   ```
   Test 1,,
   ```

2. The call must contain the minimum number of parameters as required by the called subroutine. For instance, using the above example, the following are invalid:

   ```
   Test ,1        'Only passes two out of three required
                  'parameters.
   Test 1,2       'Only passes two out of three required
                  'parameters.
   ```

When you skip a parameter in this manner, BasicScript creates a temporary variable and passes this variable instead. The value of this temporary variable depends on the data type of the corresponding parameter in the argument list of the called subroutine, as described in the following table:

Value	Data Type
0	**Integer**, **Long**, **Single**, **Double**, **Currency**
Zero-length string	**String**
Nothing	**Object** (or any data object)
Error	**Variant**
December 30, 1899	**Date**
False	**Boolean**

Within the called subroutine, you will be unable to determine whether a parameter was skipped unless the parameter was declared as a variant in the argument list of the subroutine. In this case, you can use the **IsMissing** function to determine whether the parameter was skipped:

```
Sub Test(a,b,c)
   If IsMissing(a) Or IsMissing(b) Then Exit Sub
End Sub
```

Example

```
'This example uses a subroutine to calculate the area of a circle.
Sub Main()
   r! = 10
   PrintArea r!
End Sub
Sub PrintArea(r as single)
   area! = (r! ^ 2) * Pi
   MsgBox "The area of a circle with radius " & r! & " = " & area!
End Sub
```

See Also **Main** (statement); **Function...End Function** (statement).

Platform(s) All.

Switch (function)

Syntax `Switch(`*condition1*`,`*expression1* `[,`*condition2*`,`*expression2* `...`
`[,`*condition7*`,`*expression7*`]])`

Description Returns the expression corresponding to the first **True** condition.

Comments The **Switch** function evaluates each condition and expression, returning the expression that corresponds to the first condition (starting from the left) that evaluates to **True**. Up to seven condition/expression pairs can be specified.

A runtime error is generated it there is an odd number of parameters (i.e., there is a condition without a corresponding expression).

The **Switch** function returns **Null** if no condition evaluates to **True**.

Example

```
'This code fragment displays the current operating platform. If
'the platform is unknown, then the word "Unknown" is displayed.
Sub Main()
      Dim a As Variant
      a = Switch(Basic.OS = 0,"Windows 3.1", _
         Basic.OS = 2,"Win32",Basic.OS = 11,"OS/2")
      MsgBox "The current platform is: " & _
         IIf(IsNull(a),"Unknown",a)
End Sub
```

See Also **Choose** (function); **IIf** (function); **If...Then...Else** (statement); **Select...Case** (statement).

Platform(s) All.

SYD (function)

Syntax `SYD(`*cost*`, `*salvage*`, `*life*`, `*period*`)`

Description Returns the sum of years' digits depreciation of an asset over a specific period of time.

Comments The **SYD** of an asset is found by taking an estimate of its useful life in years, assigning values to each year, and adding up all the numbers.

The formula used to find the **SYD** of an asset is as follows:

```
(Cost – Salvage_Value) * Remaining_Useful_Life / SYD
```

The **SYD** function requires the following named parameters:

Named Parameter	Description
cost	**Double** representing the initial cost of the asset.
salvage	**Double** representing the estimated value of the asset at the end of its useful life.
life	**Double** representing the length of the asset's useful life.
period	**Double** representing the period for which the depreciation is to be calculated. It cannot exceed the life of the asset.

To receive accurate results, the parameters *life* and *period* must be expressed in the same units. If *life* is expressed in terms of months, for example, then *period* must also be expressed in terms of months.

Example

```
'In this example, an asset that cost $1,000.00 is depreciated
'over ten years. The salvage value is $100.00, and the sum of
'the years' digits depreciation is shown for each year.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   For x = 1 To 10
      dep# = SYD(1000,100,10,x)
      message = message & "Year: " & x & "  Dep: " _
         & Format(dep#,"Currency") & crlf
   Next x
   MsgBox message
End Sub
```

See Also **Sln** (function); **DDB** (function).

Platform(s) All.

System.Exit (method)

Syntax `System.Exit`

Description Exits the operating environment.

Example
```
'This example asks whether the user would like to restart
'Windows after exiting.
Sub Main
   button = MsgBox("Restart Windows on exit?" _
      ,ebYesNo,"Exit Windows")
   If button = ebYes Then System.Restart  'Yes button selected.
   If button = ebNo Then System.Exit'No button selected.
End Sub
```

See Also **System.Restart** (method).

Platform(s) Windows, Win32.

System.FreeMemory (property)

Syntax `System.FreeMemory`

Description Returns a **Long** indicating the number of bytes of free memory.

Example
```
'The following example gets the free memory and converts it to
'kilobytes.
Sub Main()
   FreeMem& = System.FreeMemory
   FreeKBytes$ = Format(FreeMem& / 1000,"##,###")
   MsgBox FreeKbytes$ & " Kbytes of free memory"
End Sub
```

See Also **System.TotalMemory** (property); **System.FreeResources** (property); **Basic.FreeMemory** (property).

Platform(s) Windows, Win32.

System.FreeResources (property)

Syntax `System.FreeResources`

Description Returns an **Integer** representing the percentage of free system resources.

Comments The returned value is between 0 and 100.

Example
```
'This example gets the percentage of free resources.
```

```
Sub Main()
   FreeRes% = System.FreeResources
   MsgBox FreeRes% & "% of memory resources available."
End Sub
```

See Also **System.TotalMemory** (property); **System.FreeMemory** (property); **Basic.FreeMemory** (property).

Platform(s) Windows.

System.MouseTrails (method)

Syntax `System.MouseTrails` *isOn*

Description Toggles mouse trails on or off.

Comments If *isOn* is **True**, then mouse trails are turned on; otherwise, mouse trails are turned off.

A runtime error is generated if mouse trails is not supported on your system.

Example

```
'This example turns on mouse trails.
Sub Main
   System.MouseTrails 1
End Sub
```

Platform(s) Windows.

Platform Notes **Windows:** Under Windows, the setting is saved in the INI file permanently. Setting *isOn* to **True** restores the mouse trails setting as configured by the system (i.e., if your mouse trails is set to 4, then setting *isOn* to **True** sets the mouse trails to 4).

Win32: Under Win32, the setting is saved in the system registry. Setting *isOn* to **True** sets the mouse trails to 7. Setting *isOn* to **False** turns mouse trails off. Setting *isOn* to any value between 1 and 7 sets the mouse trails to that number of trails.

System.Restart (method)

Syntax `System.Restart`

Description Restarts the operating environment.

Example

```
'This example asks whether the user would like to restart
'Windows after exiting.
Sub Main
   button = MsgBox ("Restart Windows on exit?",ebYesNo, _
      "Exit Windows")
   If button = ebYes Then System.Restart  'Yes button selected.
```

```
   If button = ebNo Then System.Exit'No button selected.
End Sub
```

See Also **System.Exit** (method).

Platform(s) Windows, Win32.

System.TotalMemory (property)

Syntax `System.TotalMemory`

Description Returns a **Long** representing the number of bytes of available free memory in Windows.

Example

```
'This example displays the total system memory.
Sub Main()
   TotMem& = System.TotalMemory
   TotKBytes$ = Format(TotMem& / 1000,"##,###")
   MsgBox TotKbytes$ & " Kbytes of total system memory exist"
End Sub
```

See Also **System.FreeMemory** (property); **System.FreeResources** (property); **Basic.FreeMemory** (property).

Platform(s) Windows, Win32.

System.WindowsDirectory$ (property)

Syntax `System.WindowsDirectory$`

Description Returns the home directory of the operating environment.

Example

```
'This example displays the Windows directory.
Sub Main
   MsgBox "Windows directory = " & System.WindowsDirectory$
End Sub
```

See Also **Basic.HomeDir$** (property).

Platform(s) Windows, Win32.

System.WindowsVersion$ (property)

Syntax `System.WindowsVersion$`

Description Returns the version of the operating environment, such as "3.0" or "3.1."

Example

```
'This example sets the UseWin31 variable to True if the Windows
```

```
'version is greater than or equal to 3.1; otherwise, it sets the
'UseWin31 variable  to False.
Sub Main()
   If Val(System.WindowsVersion$) > 3.1 Then
      MsgBox "You are running a Windows version later than 3.1"
   Else
      MsgBox "You are running Windows version 3.1 or earlier"
   End If
End Sub
```

See Also **Basic.Version$** (property).

Platform(s) Windows, Win32.

Platform Notes Windows: Under Windows, this property returns a value such as "**3.1**" or "**3.11**".

Win32: On Win32 platforms, this property returns a value in the following format:
major.*minor*.*buildnumber*

Where *major* is the major version number, *minor* is the minor version number, and *buildnumber* is the actual build number.

Tab (function)

Syntax `Tab (`*column*`)`

Description Prints the number of spaces necessary to reach a given column position.

Comments This function can only be used with the **Print** and **Print#** statements.

The *column* parameter is an **Integer** specifying the desired column position to which to advance. It can be any value between 0 and 32767 inclusive.

Rule 1: If the current print position is less than or equal to *column*, then the number of spaces is calculated as:

```
column – print_position
```

Rule 2: If the current print position is greater than *column*, then *column* – 1 spaces are printed on the next line.

If a line width is specified (using the **Width** statement), then the column position is adjusted as follows before applying the above two rules:

```
column = column Mod width
```

The **Tab** function is useful for making sure that output begins at a given column position, regardless of the length of the data already printed on that line.

Example

```
'This example prints three column headers and three numbers
'aligned below the column headers.
Sub Main()
   Viewport.Open
   Print "Column1";Tab(10);"Column2";Tab(20);"Column3"
   Print Tab(3);"1";Tab(14);"2";Tab(24);"3"
   Sleep(10000)         'Wait 10 seconds.
   Viewport.Close
End Sub
```

See Also **Spc** (function); **Print** (statement); **Print#** (statement).

Platform(s) All.

Tan (function)

Syntax `Tan(`*number*`)`

Description Returns a **Double** representing the tangent of *number*.

Comments The *number* parameter is a **Double** value given in radians.

Example

```
'This example computes the tangent of pi/4 radians (45 degrees).
Sub Main()
```

```
   c# = Tan(Pi / 4)
   MsgBox "The tangent of 45 degrees is: " & c#
End Sub
```

See Also **Sin** (function); **Cos** (function); **Atn** (function).

Platform(s) All.

Text (statement)

Syntax `Text` *x*,*y*,*width*,*height*,*title$* `[,[.`*Identifier*`] [,[`*FontName$*`] [,[`*size*`] [,`*style*`]]]]`

Description Defines a text control within a dialog box template. The text control only displays text; the user cannot set the focus to a text control or otherwise interact with it.

Comments The text within a text control word-wraps. Text controls can be used to display up to 32K of text.

The **Text** statement accepts the following parameters:

Parameter	Description
x, *y*	**Integer** positions of the control (in dialog units) relative to the upper left corner of the dialog box.
width, *height*	**Integer** dimensions of the control in dialog units.
title$	**String** containing the text that appears within the text control. This text may contain an ampersand character to denote an accelerator letter, such as "&Save" for Save. Pressing this accelerator letter sets the focus to the control following the **Text** statement in the dialog box template.
.*Identifier*	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). If this parameter is omitted, then the first two words from *title$* are used.
FontName$	Name of the font used for display of the text within the text control. If this parameter is omitted, then the default font for the dialog is used.
size	Size of the font used for display of the text within the text control. If this parameter is omitted, then the default size for the default font of the dialog is used.

Parameter	Description
style	Style of the font used for display of the text within the text control. This can be any of the following values: **ebRegular** Normal font (i.e., neither bold nor italic) **ebBold** Bold font **ebItalic** Italic font **ebBoldItalic** Bold-italic font If this parameter is omitted, then **ebRegular** is used.

Example

```
Begin Dialog UserDialog3 81,64,128,60,"Untitled"
   CancelButton 80,32,40,14
   OKButton 80,8,40,14
   Text 4,8,68,44,"This text is displayed in the dialog box."
End Dialog
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **TextBox** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Platform Notes **Windows, Win32:** Under Windows and Win32, accelerators are underlined, and the Alt+*letter* accelerator combination is used.

OS/2: Under OS/2, accelerators are underlined, and the Alt+*letter* accelerator combination is used.

Macintosh: On the Macintosh, accelerators are normal in appearance, and the Command+*letter* accelerator combination is used.

TextBox (statement)

Syntax `TextBox` *x*`,`*y*`,`*width*`,`*height*`,`*.Identifier* `[,[`*isMultiline*`] [,[`*FontName$*`] [,[`*size*`] [,`*style*`]]]]`

Description Defines a single or multiline text-entry field within a dialog box template.

Comments

The **TextBox** statement requires the following parameters:

Parameter	Description
x, y	**Integer** position of the control (in dialog units) relative to the upper left corner of the dialog box.
width, height	**Integer** dimensions of the control in dialog units.
.Identifier	Name by which this control can be referenced by statements in a dialog function (such as **DlgFocus** and **DlgEnable**). This parameter also creates a string variable whose value corresponds to the content of the text box. This variable can be accessed using the syntax: *DialogVariable.Identifier*
isMultiline	Specifies whether the text box can contain more than a single line (0 = single-line; 1 = multiline).
FontName$	Name of the font used for display of the text within the text box control. If this parameter is omitted, then the default font for the dialog is used.
size	Size of the font used for display of the text within the text box control. If this parameter is omitted, then the default size for the default font of the dialog is used.
style	Style of the font used for display of the text within the text box control. This can be any of the following values: **ebRegular** Normal font (i.e., neither bold nor italic) **ebBold** Bold font **ebItalic** Italic font **ebBoldItalic** Bold-italic font If this parameter is omitted, then **ebRegular** is used.

If *isMultiline* is 1, the **TextBox** statement creates a multiline text-entry field. When the user types into a multiline field, pressing the Enter key creates a new line rather than selecting the default button.

The *isMultiLine* parameter also specifies whether the text box is read-only and whether the text-box should hide input for password entry. To specify these extra parameters, you can form the isMultiLine parameter by ORing together the following values:

Value	Meaning
0	Text box is single-line.
1	Text box is multi-line.

Value	Meaning
&H8000	Text box is read-only.
&H4000	Text box is password-entry.

For example, the following statement creates a read-only multiline text box:

```
TextBox 10,10,80,14,.TextBox1,1 Or &H8000
```

The **TextBox** statement can only appear within a dialog box template (i.e., between the **Begin Dialog** and **End Dialog** statements).

When the dialog box is created, the *.Identifier* variable is used to set the initial content of the text box. When the dialog box is dismissed, the variable will contain the new content of the text box.

A single-line text box can contain up to 256 characters. The length of text in a multiline text box is not limited by BasicScript; the default memory limit specified by the given platform is used instead.

Example

```
Begin Dialog UserDialog3 81,64,128,60,"Untitled"
   CancelButton 80,32,40,14
   OKButton 80,8,40,14
   TextBox 4,8,68,44,.TextBox1,1
End Dialog
```

See Also **CancelButton** (statement); **CheckBox** (statement); **ComboBox** (statement); **Dialog** (function); **Dialog** (statement); **DropListBox** (statement); **GroupBox** (statement); **ListBox** (statement); **OKButton** (statement); **OptionButton** (statement); **OptionGroup** (statement); **Picture** (statement); **PushButton** (statement); **Text** (statement); **Begin Dialog** (statement); **PictureButton** (statement); **HelpButton** (statement).

Platform(s) Windows, Win32, Macintosh, OS/2, UNIX.

Time, Time$ (functions)

Syntax

```
Time[$][()]
```

Description Returns the system time as a **String** or as a **Date** variant.

Comments The **Time$** function returns a string that contains the time in a 24-hour time format, whereas **Time** returns a **Date** variant.

To set the time, use the **Time/Time$** statements.

Example

```
'This example returns the system time and displays it in a
'dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
```

```
   oldtime$ = Time$
   message = "Time was: " & oldtime$ & crlf
   Time$ = "10:30:54"
   message = message & "Time set to: " & Time$ & crlf
   Time$ = oldtime$
   message = message & "Time restored to: " & Time$
   MsgBox message
End Sub
```

See Also **Time**, **Time$** (statements); **Date**, **Date$** (functions); **Date**, **Date$** (statements); **Now** (function).

Platform(s) All.

Time, Time$ (statements)

Syntax `Time[$] = newtime`

Description Sets the system time to the time contained in the specified string.

Comments The **Time$** statement requres a string variable in one of the following formats:

HH

HH:MM

HH:MM:SS

where *HH* is between 0 and 23, *MM* is between 0 and 59, and *SS* is between 0 and 59.

The **Time** statement converts any valid expression to a time, including string and numeric values. Unlike the **Time$** statement, **Time** recognizes many different time formats, including 12-hour times.

Example

```
'This example returns the system time and displays it in a
'dialog box.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   oldtime$ = Time$
   message = "Time was: " & oldtime$ & crlf
   Time$ = "10:30:54"
   message = message & "Time set to: " & Time$ & crlf
   Time$ = oldtime$
   message = message & "Time restored to: " & Time$
   MsgBox message
End Sub
```

See Also **Time**, **Time$** (functions); **Date**, **Date$** (functions); **Date**, **Date$** (statements).

Platform(s) All.

Platform Notes **UNIX, Win32, OS/2:** On all UNIX platforms, Win32, and OS/2, you may not have permission to change the time, causing runtime error 70 to be generated.

Timer (function)

Syntax `Timer`

Description Returns a **Single** representing the number of seconds that have elapsed since midnight.

Example

```
'This example displays the elapsed time between execution start
'and the time you clicked the OK button on the first message.
Sub Main()
   start& = Timer
   MsgBox "Click the OK button, please."
   total& = Timer - start&
   MsgBox "The elapsed time was: " & total& & " seconds."
End Sub
```

See Also **Time**, **Time$** (functions); **Now** (function).

Platform(s) All.

TimeSerial (function)

Syntax `TimeSerial(`*hour*`, `*minute*`, `*second*`)`

Description Returns a **Date** variant representing the given time with a date of zero.

Comments The **TimeSerial** function requires the following named parameters:

Named Parameter	Description
hour	**Integer** between 0 and 23.
minute	**Integer** between 0 and 59.
second	**Integer** between 0 and 59.

Example

```
Sub Main()
   start# = TimeSerial(10,22,30)
   finish# = TimeSerial(10,35,27)
   dif# = Abs(start# - finish#)
   MsgBox "The time difference is: " & Format(dif#, "hh:mm:ss")
End Sub
```

See Also **DateValue** (function); **TimeValue** (function); **DateSerial** (function).

Platform(s) All.

TimeValue (function)

Syntax `TimeValue(`*time*`)`

Description Returns a **Date** variant representing the time contained in the specified string argument.

Comments This function interprets the passed *time* parameter looking for a valid time specification.

The *time* parameter can contain valid time items separated by time separators such as colon (:) or period (.).

Time strings can contain an optional date specification, but this is not used in the formation of the returned value.

If a particular time item is missing, then it is set to 0. For example, the string "10 pm" would be interpreted as "22:00:00."

Example

```
'This example calculates the current time and displays it in a
'dialog box.
Sub Main()
   t1$ = "10:15"
   t2# = TimeValue(t1$)
   MsgBox "The TimeValue of " & t1$ & " is: " & t2#
End Sub
```

See Also **DateValue** (function); **TimeSerial** (function); **DateSerial** (function).

Platform(s) All.

Platform Notes **Windows:** Under Windows, time specifications vary, depending on the international settings contained in the [intl] section of the win.ini file.

Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$ (functions)

Syntax

```
Trim[$](string)
LTrim[$](string)
RTrim[$](string)
```

Description Returns a copy of the passed string expression (*string*) with leading and/or trailing spaces removed.

Comments **Trim** returns a copy of the passed string expression (*string*) with both the leading and trailing spaces removed. **LTrim** returns *string* with the leading spaces removed, and **RTrim** returns *string* with the trailing spaces removed.

Trim$**, **LTrim$, and **RTrim$** return a **String**, whereas **Trim**, **LTrim**, and **RTrim** return a **String** variant.

Null is returned if *string* is **Null**.

Examples

```
'This first example uses the Trim$ function to extract the
'nonblank part of a string and display it.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   text$ = "      This is text           "
   tr$ = Trim$(text$)
   MsgBox "Original =>" & text$ & "<=" & crlf & _
      "Trimmed =>" & tr$ & "<="
End Sub
'This second example displays a right-justified string and its
'LTrim result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "          <= This is a right-justified string"
   b$ = LTrim$(a$)
   MsgBox a$ & crlf & b$
End Sub
'This third example displays a left-justified string and its
'RTrim result.
Const crlf = Chr$(13) + Chr$(10)
Sub Main()
   a$ = "This is a left-justified string.             "
   b$ = RTrim$(a$)
   MsgBox a$ & "<=" & crlf & b$ & "<="
End Sub
```

Platform(s) All.

Type (statement)

Syntax

```
Type username
   variable As type
   variable As type
   variable As type
   :
End Type
```

Description The **Type** statement creates a structure definition that can then be used with the **Dim** statement to declare variables of that type. The *username* field specifies the name of the structure that is used later with the **Dim** statement.

Comments Within a structure definition appear field descriptions in the format:

variable As *type*

where *variable* is the name of a field of the structure, and *type* is the data type for that variable. Any fundamental data type or previously declared user-defined data type can be used within the structure definition (structures within structures are allowed). Only fixed arrays can appear within structure definitions.

The **Type** statement can only appear outside of subroutine and function declarations.

When declaring strings within fixed-size types, it is useful to declare the strings as fixed-length. Fixed-length strings are stored within the structure itself rather than in the string space. For example, the following structure will always require 62 bytes of storage:

```
Type Person
   FirstName As String * 20
   LastName As String * 40
   Age As Integer
End Type
```

Note: Fixed-length strings within structures are size-adjusted upward to an even byte boundary. Thus, a fixed-length string of length 5 will occupy 6 bytes of storage within the structure.

Example

```
'This example displays the use of the Type statement to create
'a structure representing the parts of a circle and assign
'values to them.
Type Circ
   message As String
   rad As Integer
   dia As Integer
   are As Double
   cir As Double
End Type
Sub Main()
   Dim circle As Circ
   circle.rad = 5
   circle.dia = circle.rad * 2
   circle.are = (circle.rad ^ 2) * Pi
   circle.cir = circle.dia * Pi
   circle.message = "The area of the circle is: " & circle.are
   MsgBox circle.message
End Sub
```

See Also **Dim** (statement); **Public** (statement); **Private** (statement).

Platform(s) All.

TypeName (function)

Syntax `TypeName(`*varname*`)`

Description Returns the type name of the specified variable.

Comments The returned string can be any of the following:

Returned String	Returned if varname is
"String"	A String.
objecttype	A data object variable. In this case, *objecttype* is the name of the specific object type.
"Integer"	An integer.
"Long"	A long.
"Single"	A single.
"Double"	A double.
"Currency"	A currency value.
"Date"	A date value.
"Boolean"	A boolean value.
"Error"	An error value.
"Empty"	An uninitialized variable.
"Null"	A variant containing no valid data.
"Object"	An OLE automation object.
"Unknown"	An unknown type of OLE automation object.
"Nothing"	An uninitialized object variable.
class	A specific type of OLE automation object. In this case, *class* is the name of the object as known to OLE.

If *varname* is an array, then the returned string can be any of the above strings follows by a empty parenthesis. For example, **"Integer()"** would be returned for an array of integers.

If *varname* is an expression, then the expression is evaluated and a **String** representing the resultant data type is returned.

If *varname* is an OLE collection, then **TypeName** returns the name of that object collection.

Example

```
'The following example defines a subroutine that only accepts
'Integer variables. If not passed an Integer, it will inform
'the user that there was an error, displaying the actual type
'of variable that was passed.
```

```
Sub Foo(a As Variant)
   If VarType(a) <> ebInteger Then
      MsgBox "Foo does not support " & TypeName(a) & " variables"
   End If
End Sub
```

See Also **TypeOf** (function).

Platform(s) All.

TypeOf (function)

Syntax `TypeOf` *objectvariable* `Is` *objecttype*

Description Returns **True** if *objectvariable* the specified typel **False** otherwise.

Comments This function is used within the **If...Then** statement to determine if a variable is of a particular type. This function is particularily useful for determining the type of OLE automation objects.

Example

```
Sub Main()
   Dim a As Object
   Set a = CreateObject("Excel.Application")
   If TypeOf a Is "Application" Then
      MsgBox "We have an Application object."
   End If
End Sub
```

See Also **TypeName** (function).

Platform(s) All.

UBound (function)

Syntax `UBound(`*ArrayVariable*`() [,`*dimension*`])`

Description Returns an **Integer** containing the upper bound of the specified dimension of the specified array variable.

Comments The *dimension* parameter is an integer that specifies the desired dimension. If not specified, then the upper bound of the first dimension is returned.

The **UBound** function can be used to find the upper bound of a dimension of an array returned by an OLE Automation method or property:

`UBound``(`*object.property* `[,`*dimension*`])`

`UBound``(`*object.method* `[,`*dimension*`])`

Examples

```
'This example dimensions two arrays and displays their upper
```

```
'bounds.
Sub Main()
   Dim a(5 To 12)
   Dim b(2 To 100, 9 To 20)
   uba = UBound(a)
   ubb = UBound(b,2)
   MsgBox "The upper bound of a is: " & uba & " The upper bound of
b is: " & ubb
'This example uses Lbound and Ubound to dimension a dynamic
'array to hold a copy of an array redimmed by the FileList
'statement.
Dim fl$()
FileList fl$,"*"
count = Ubound(fl$)
If ArrayDims(a) Then
   Redim nl$(Lbound(fl$) To Ubound(fl$))
   For x = 1 To count
      nl$(x) = fl$(x)
   Next x
   MsgBox "The last element of the new array is: " & nl$(count)
   End If
End Sub
```

See Also **LBound** (function); **ArrayDims** (function); Arrays (topic).

Platform(s) All.

UCase, UCase$ (functions)

Syntax `UCase[$](`*string*`)`

Description Returns the uppercase equivalent of the specified string.

Comments **UCase$** returns a **String**, whereas **UCase** returns a **String** variant.

Null is returned if *string* is **Null**.

Example

```
'This example uses the UCase$ function to change a string from
'lowercase to uppercase.
Sub Main()
   a1$ = "this string was lowercase, but was converted."
   a2$ = UCase$(a1$)
   MsgBox a2$
End Sub
```

See Also **LCase**, **LCase$** (functions).

Platform(s) All.

Unlock (statement)

See **Lock**, **Unlock** (statements).

User-Defined Types (topic)

User-defined types (UDTs) are structure definitions created using the **Type** statement. UDTs are equivalent to C language structures.

Declaring Structures

The **Type** statement is used to create a structure definition. Type declarations must appear outside the body of all subroutines and functions within a script and are therefore global to an entire script.

Once defined, a UDT can be used to declare variables of that type using the **Dim**, **Public**, or **Private** statement. The following example defines a rectangle structure:

```
Type Rect
   left As Integer
   top As Integer
   right As Integer
   bottom As Integer
End Type
   :
Sub Main()
   Dim r As Rect
      :
      r.left = 10
   End Sub
```

Any fundamental data type can be used as a structure member, including other user-defined types. Only fixed arrays can be used within structures.

Copying Structures

UDTs of the same type can be assigned to each other, copying the contents. No other standard operators can be applied to UDTs.

```
Dim r1 As Rect
Dim r2 As Rect
   :
r1 = r2
```

When copying structures of the same type, all strings in the source UDT are duplicated and references are placed into the target UDT.

The **LSet** statement can be used to copy a UDT variable of one type to another:

```
LSet variable1 = variable2
```

LSet cannot be used with UDTs containing variable-length strings. The smaller of the two structures determines how many bytes get copied.

Passing Structures

UDTs can be passed both to user-defined routines and to external routines, and they can be assigned. UDTs are always passed by reference.

Since structures are always passed by reference, the **ByVal** keyword cannot be used when defining structure arguments passed to external routines (using **Declare**). The **ByVal** keyword can only be used with fundamental data types such as **Integer** and **String**.

Passing structures to external routines actually passes a far pointer to the data structure.

Size of Structures

The **Len** function can be used to determine the number of bytes occupied by a UDT:

```
Len(udt_variable_name)
```

Since strings are stored in BasicScript's data space, only a reference (currently, 2 bytes) is stored within a structure. Thus, the **Len** function may seem to return incorrect information for structures containing strings.

Val (function)

Syntax `Val(`*string*`)`

Description Converts a given string expression to a number.

Comments The *string* parameter can contain any of the following:

- Leading minus sign (for nonhex or octal numbers only)
- Hexadecimal number in the format &H*hexdigits*
- Octal number in the format &O*octaldigits*
- Floating-point number, which can contain a decimal point and an optional exponent

Spaces, tabs, and line feeds are ignored.

If *string* does not contain a number, then 0 is returned.

The **Val** function continues to read characters from the string up to the first nonnumeric character.

The **Val** function always returns a double-precision floating-point value. This value is forced to the data type of the assigned variable.

Example

```
'This example inputs a number string from an InputBox and
'converts it to a number variable.
Sub Main()
   a$ = InputBox$("Enter anything containing a number", _
      "Enter Number")
   b# = Val(a$)
   MsgBox "The value is: " & b#
End Sub
```

See Also **CDbl** (function); **Str**, **Str$** (functions).

Platform(s) All.

Variant (data type)

Syntax `Variant`

Description A data type used to declare variables that can hold one of many different types of data.

Comments During a variant's existence, the type of data contained within it can change. Variants can contain any of the following types of data:

Type of Data	BasicScript Data Types
Numeric	**Integer**, **Long**, **Single**, **Double**, **Boolean**, **Date**, **Currency**.
Logical	**Boolean**.
Dates and times	**Date**.
String	**String**.
Object	**Object**.
No valid data	A variant with no valid data is considered **Null**.
Uninitialized	An uninitialized variant is considered **Empty**.

There is no type-declaration character for variants.

The number of significant digits representable by a variant depends on the type of data contained within the variant.

Variant is the default data type for BasicScript. If a variable is not explicitly declared with **Dim**, **Public**, or **Private**, and there is no type-declaration character (i.e., #, @, !, %, or &), then the variable is assumed to be **Variant**.

Determining the Subtype of a Variant

The following functions are used to query the type of data contained within a variant:

Function	Description
VarType	Returns a number representing the type of data contained within the variant.
IsNumeric	Returns **True** if a variant contains numeric data. The following are considered numeric: **Integer**, **Long**, **Single**, **Double**, **Date**, **Boolean**, **Currency** If a variant contains a string, this function returns **True** if the string can be converted to a number. If a variant contains an **Object** whose default property is numeric, then **IsNumeric** returns **True**.
IsObject	Returns **True** if a variant contains an object.
IsNull	Returns **True** if a variant contains no valid data.
IsEmpty	Returns **True** if a variant is uninitialized.
IsDate	Returns **True** if a variant contains a date. If the variant contains a string, then this function returns **True** if the string can be converted to a date. If the variant contains an **Object**, then this function returns **True** if the default property of that object can be converted to a date.

Assigning to Variants

Before a **Variant** has been assigned a value, it is considered empty. Thus, immediately after declaration, the **VarType** function will return **ebEmpty**. An uninitialized variant is 0 when used in numeric expressions and is a zero-length string when used within string expressions.

A **Variant** is **Empty** only after declaration and before assigning it a value. The only way for a **Variant** to become **Empty** after having received a value is for that variant to be assigned to another **Variant** containing **Empty**, for it to be assigned explicitly to the constant **Empty**, or for it to be erased using the **Erase** statement.

When a variant is assigned a value, it is also assigned that value's type. Thus, in all subsequent operations involving that variant, the variant will behave like the type of data it contains.

Operations on Variants

Normally, a **Variant** behaves just like the data it contains. One exception to this rule is that, in arithmetic operations, variants are automatically promoted when an overflow occurs. Consider the following statements:

```
Dim a As Integer,b As Integer,c As Integer
Dim x As Variant,y As Variant,z As Variant
a% = 32767
b% = 1
c% = a% + b%   'This will overflow.
x = 32767
y = 1
z = x + y      'z becomes a Long because of Integer
               'overflow.
```

In the above example, the addition involving **Integer** variables overflows because the result (32768) overflows the legal range for integers. With **Variant** variables, on the other hand, the addition operator recognizes the overflow and automatically promotes the result to a **Long**.

Adding Variants

The + operator is defined as performing two functions: when passed strings, it concatenates them; when passed numbers, it adds the numbers.

With variants, the rules are complicated because the types of the variants are not known until execution time. If you use +, you may unintentionally perform the wrong operation.

It is recommended that you use the **&** operator if you intend to concatenate two **String** variants. This guarantees that string concatenation will be performed and not addition.

Variants That Contain No Data

A **Variant** can be set to a special value indicating that it contains no valid data by assigning the **Variant** to **Null**:

```
Dim a As Variant
a = Null
```

The only way that a **Variant** becomes **Null** is if you assign it as shown above.

The **Null** value can be useful for catching errors since its value propagates through an expression.

Variant Storage

Variants require 16 bytes of storage internally:

- A 2-byte type
- A 2-byte extended type for data objects
- 4 bytes of padding for alignment
- An 8-byte value

Unlike other data types, writing variants to **Binary** or **Random** files does not write 16 bytes. With variants, a 2-byte type is written, followed by the data (2 bytes for **Integer** and so on).

Disadvantages of Variants

The following list describes some disadvantages of variants:

1. Using variants is slower than using the other fundamental data types (i.e., **Integer**, **Long**, **Single**, **Double**, **Date**, **Object**, **String**, **Currency**, and **Boolean**). Each operation involving a **Variant** requires examination of the variant's type.
2. Variants require more storage than other data types (16 bytes as opposed to 8 bytes for a **Double**, 2 bytes for an **Integer**, and so on).
3. Unpredictable behavior. You may write code to expect an **Integer** variant. At runtime, the variant may be automatically promoted to a **Long** variant, causing your code to break.

Passing Nonvariant Data to Routines Taking Variants

Passing nonvariant data to a routine that is declared to receive a variant by reference prevents that variant from changing type within that routine. For example:

```
Sub Foo(v As Variant)
   v = 50                  'OK.
   v = "Hello, world."     'Get a type-mismatch error here!
End Sub
Sub Main()
   Dim i As Integer
   Foo i                   'Pass an integer by reference.
End Sub
```

In the above example, since an **Integer** is passed by reference (meaning that the caller can change the original value of the **Integer**), the caller must ensure that no attempt is made to change the variant's type.

Passing Variants to Routines Taking Nonvariants

Variant variables cannot be passed to routines that accept nonvariant data by reference, as demonstrated in the following example:

```
Sub Foo(i as Integer)
End Sub
Sub Main()
   Dim a As Variant
   Foo a             'Compiler gives type-mismatch error here.
End Sub
```

See Also **Currency** (data type); **Date** (data type); **Double** (data type); **Integer** (data type); **Long** (data type); **Object** (data type); **Single** (data type); **String** (data type); **Boolean** (data type); **Def***Type* (statement); **CVar** (function); **VarType** (function).

Platform(s) All.

VarType (function)

Syntax `VarType(`*varname*`)`

Description Returns an **Integer** representing the type of data in *varname*.

Comments The *varname* parameter is the name of any **Variant**.

The following table shows the different values that can be returned by **VarType**:

Value	Constant	Data Type
0	**ebEmpty**	Uninitialized
1	**ebNull**	No valid data
2	**ebInteger**	**Integer**
3	**ebLong**	**Long**
4	**ebSingle**	**Single**
5	**ebDouble**	**Double**
6	**ebCurrency**	**Currency**
7	**ebDate**	**Date**
8	**ebString**	**String**
9	**ebObject**	**Object** (OLE Automation object)
10	**ebError**	User-defined error
11	**ebBoolean**	**Boolean**
12	**ebVariant**	**Variant** (not returned by this function)
13	**ebDataObject**	Non–OLE Automation object

When passed an object, the **VarType** function returns the type of the default property of that object. If the object has no default property, then either **ebObject** or **ebDataObject** is returned, depending on the type of variable.

Example

```
Sub Main()
   Dim v As Variant
   v = 5&         'Set v to a Long.
   If VarType(v) = ebInteger Then
      Msgbox "v is an Integer."
   ElseIf VarType(v) = ebLong Then
```

```
      Msgbox "v is a Long."
   End If
End Sub
```

See Also **Variant** (data type).

Platform(s) All.

Viewport.Clear (method)

Syntax `Viewport.Clear`

Description Clears the open viewport window.

Comments The method has no effect if no viewport is open.

Example

```
Sub Main()
   Viewport.Open
   Print "This will be displayed in the viewport window."
   Sleep 2000
   Viewport.Clear
   Print "This will replace the previous text."
   Sleep 2000
   Viewport.Close
End Sub
```

See Also **Viewport.Close** (method); **Viewport.Open** (method).

Platform(s) Windows, Win32.

Viewport.Close (method)

Syntax `Viewport.Close`

Description This method closes an open viewport window.

Comments The method has no effect if no viewport is opened.

Example

```
Sub Main()
   Viewport.Open
   Print "This will be displayed in the viewport window."
   Sleep 2000
   Viewport.Close
End Sub
```

See Also **Viewport.Open** (method).

Platform(s) Windows, Win32.

Viewport.Open (method)

Syntax `Viewport.Open [`*title* `[,`*XPos*`,`*YPos* `[,`*width*`,`*height*`]]]`

Description Opens a new viewport window or switches the focus to the existing viewport window.

Comments The **Viewport.Open** method accepts the following named :

Named Parameter	Description
title	Specifies a **String** containing the text to appear in the viewport's caption.
XPos, YPos	Specifies **Integer** coordinates given in twips indicating the initial position of the upper left corner of the viewport.
width,height	Specifies **Integer** values indicating the initial width and height of the viewport.

If a viewport window is already open, then it is given the focus. Otherwise, a new viewport window is created.

Combined with the **Print** statement, a viewport window is a convenient place to output debugging information.

The viewport window is closed when the BasicScript host application is terminated.

The following keys work within a viewport window:

Up	Scrolls up by one line.
Down	Scrolls down by one line.
Home	Scrolls to the first line in the viewport window.
End	Scrolls to the last line in the viewport window.
PgDn	Scrolls the viewport window down by one page.
PgUp	Scrolls the viewport window up by one page.
Ctrl+PgUp	Scrolls the viewport window left by one page.
Ctrl+PgDn	Scrolls the viewport window right by one page.

Only one viewport window can be open at any given time. Any scripts with **Print** statements will output information into the same viewport window.

When printing to viewports, the end-of-line character can be any of the following: a carriage return, a line feed, or a carriage-return/line-feed pair. Embedded null characters are printed as spaces.

Example

```
Sub Main()
   Viewport.Open "BasicScript Viewport",100,100,500,500
   Print "This will be displayed in the viewport window."
```

```
   Sleep 2000
   Viewport.Close
End Sub
```

See Also **Viewport.Close** (method).

Platform(s) Windows, Win32.

Platform Notes **Windows:** The buffer size for the viewport is 32K. Information from the start of the buffer is removed to make room for additional information being appended to the end of the buffer.

VLine (statement)

Syntax `VLine [`*lines*`]`

Description Scrolls the window with the focus up or down by the specified number of lines.

Comments The *lines* parameter is an **Integer** specifying the number of lines to scroll. If this parameter is omitted, then the window is scrolled down by one line.

Example

```
'This example prints a series of lines to the viewport, then
'scrolls back up the lines to the top using VLine.
Sub Main()
   Viewport.Open "BasicScript Viewport",100,100,500,200
   For i = 1 to 50
      Print "This will be displayed on line#: " & i
   Next i
   MsgBox "We will now go back 40 lines..."
   VLine -40
   MsgBox "...and here we are!"
   Viewport.Close
End Sub
```

See Also **VPage** (statement); **VScroll** (statement).

Platform(s) Windows, Win32.

VPage (statement)

Syntax `VPage [`*pages*`]`

Description Scrolls the window with the focus up or down by the specified number of pages.

Comments The *pages* parameter is an **Integer** specifying the number of lines to scroll. If this parameter is omitted, then the window is scrolled down by one page.

Example

```
'This example scrolls the viewport window up five pages.
```

```
Sub Main()
   Viewport.Open "BasicScript Viewport",100,100,500,200
   For i = 1 to 500
      Print "This will be displayed on line#: " & i
   Next i
   MsgBox "We will now go back 5 pages..."
   VLine -5
   MsgBox "...and here we are!"
   Viewport.Close
End Sub
```

See Also **VLine** (statement); **VScroll** (statement).

Platform(s) Windows, Win32.

VScroll (statement)

Syntax `VScroll` *percentage*

Description Sets the thumb mark on the vertical scroll bar attached to the current window.

Comments The position is given as a percentage of the total range associated with that scroll bar. For example, if the percentage parameter is 50, then the thumb mark is positioned in the middle of the scroll bar.

Example

```
'This example prints a bunch of lines to the viewport, then
'scrolls back to the top using VScroll.
Sub Main()
   Viewport.Open "BasicScript Viewport",100,100,500,200
   For i = 1 to 50
      Print "This will be displayed on line#: " & i
   Next i
   MsgBox "We will now go to the 0% thumb mark poisiton (the
top)..."
   VScroll 0
   MsgBox "...and here we are!"
   Viewport.Close
End Sub
```

See Also **VLine** (statement); **VPage** (statement).

Platform(s) Windows, Win32.

Weekday (function)

Syntax `Weekday(`*date* `[,`*firstdayofweek*`])`

Description

Returns an **Integer** value representing the day of the week given by date. Sunday is 1, Monday is 2, and so on.

The **Weekday** function takes the following named parameters:

Named Parameter	Description
date	Any expression representing a valid date.
firstdayofweek	Indicates the first day of the week. If omitted, then sunday is assumed (i.e., the constant **ebSunday** described below).

The *firstdayofweek* parameter, if specified, can be any of the following constants:

Constant	Value	Description
ebUseSystem	0	Use the system setting for *firstdayofweek*.
ebSunday	1	Sunday (the default)
ebMonday	2	Monday
ebTuesday	3	Tuesday
ebWednesday	4	Wednesday
ebThursday	5	Thursday
ebFriday	6	Friday
ebSaturday	7	Saturday

Example

```
'This example gets a date in an input box and displays the day
'of the week and its name for the date entered.
Sub Main()
   Dim a$(7)
   a$(1) = "Sunday"
   a$(2) = "Monday"
   a$(3) = "Tuesday"
   a$(4) = "Wednesday"
   a$(5) = "Thursday"
   a$(6) = "Friday"
   a$(7) = "Saturday"
Reprompt:
   bd = InputBox$("Please enter your birthday.","Enter Birthday")
   If Not(IsDate(bd)) Then Goto Reprompt
   dt = DateValue(bd)
   dw = WeekDay(dt)
   Msgbox "You were born on day " & dw & ", which was a " & a$(dw)
End Sub
```

See Also

Day (function); **Minute** (function); **Second** (function); **Month** (function); **Year** (function); **Hour** (function); **DatePart** (function).

Platform(s)

All.

While...Wend (statement)

Syntax

```
While condition
  [statements]
Wend
```

Description Repeats a statement or group of statements while a condition is **True**.

Comments The condition is initially and then checked at the top of each iteration through the loop.

Example

```
'This example executes a While loop until the random number
'generator returns a value of 1.
Sub Main()
   x% = 0
   count% = 0
   While x% <> 1 And count% < 500
      x% = Rnd(1)
      If count% > 1000 Then
         Exit Sub
      Else
         count% = count% + 1
      End If
   Wend
   MsgBox "The loop executed " & count% & " times."
End Sub
```

See Also **Do...Loop** (statement); **For...Next** (statement).

Platform(s) All.

Platform Notes **Windows, Win32:** Due to errors in program logic, you can inadvertantly create infinite loops in your code. Under Windows and Win32, you can break out of infinite loops using Ctrl+Break.

UNIX: Due to errors in program logic, you can inadvertantly create infinite loops in your code. Under UNIX, you can break out of infinite loops using Ctrl+C.

Macintosh: Due to errors in program logic, you can inadvertantly create infinite loops in your code. On the Macintosh, you can break out of infinite loops using Command+Period.

OS/2: Due to errors in program logic, you can inadvertantly create infinite loops in your code. Under OS/2, you can break out of infinite loops using Ctrl+C or Ctrl+Break.

Width# (statement)

Syntax

```
Width# filenumber, width
```

Description Specifies the line width for sequential files opened in either **Output** or **Append** mode.

Comments The **Width#** statement requires the following named parameters:

Named Parameter	Description
filenumber	**Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.
width	**Integer** between 0 to 255 inclusive specifying the new width. If *width* is 0, then no maximum line length is used.

When a file is initially opened, there is no limit to line length. This command forces all subsequent output to the specified file to use the specified value as the maximum line length.

The **Width** statement affects output in the following manner: if the column position is greater than 1 and the length of the text to be written to the file causes the column position to exceed the current line width, then the data is written on the next line.

The **Width** statement also affects output of the **Print** command when used with the **Tab** and **Spc** functions.

Example

```
'This statement sets the maximum line width for file number 1
'to 80 columns.
Sub Main()
   Width #1,80
End Sub
```

See Also **Print** (statement); **Print#** (statement); **Tab** (function); **Spc** (function).

Platform(s) All.

WinActivate (statement)

Syntax `WinActivate [`*window_name$* `|` *window_object*`] [,`*timeout*`]`

Description Activates the window with the given name or object value.

Comments The **WinActivate** statement requires the following parameters:

Parameter	Description
window_name$	String containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.
timeout	**Integer** specifying the number of milliseconds for which to attempt activation of the specified window. If not specified (or 0), then only one attempt will be made to activate the window. This value is handy when you are not certain that the window you are attempting to activate has been created.

If *window_name$* and *window_object* are omitted, then no action is performed.

Example

```
'This example runs the clock.exe program by activating the Run
'File  dialog box from within Program Manager.
Sub Main()
   WinActivate "Program Manager"
   Menu "File.Run"
   WinActivate "Program Manager|Run"
   SendKeys "clock.exe{ENTER}"
End Sub
```

See Also **AppActivate** (statement).

Platform(s) Windows, Win32.

WinClose (statement)

Syntax `WinClose [`*window_name$* `|` *window_object*`]`

Description Closes the given window.

Comments The **WinClose** statement requires the following parameters:

Parameter	Description
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is closed.

This command differs from the **AppClose** command in that this command operates on the current window rather than the current top-level window (or application).

Example

```
'This example closes Microsoft Word if its object reference is
'found.
Sub Main()
   Dim WordHandle As HWND
   Set WordHandle = WinFind("Word")
   If (WordHandle Is Not Nothing) Then WinClose WordHandle
End Sub
```

See Also **WinFind** (function).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

WinFind (function)

Syntax `WinFind(`*name$*`) As HWND`

Description Returns an object variable referencing the window having the given name.

Comments The *name$* parameter is specified using the same format as that used by the **WinActivate** statement.

Example
```
'This example closes Microsoft Word if its object reference is
'found.
Sub Main()
   Dim WordHandle As HWND
   Set WordHandle = WinFind("Word")
   If (WordHandle Is Not Nothing) Then WinClose WordHandle
End Sub
```

See Also **WinActivate** (statement).

Platform(s) Windows, Win32.

WinList (statement)

Syntax `WinList` *ArrayOfWindows*`()`

Description Fills the passed array with references to all the top-level windows.

Comments The passed array must be declared as an array of **HWND** objects.

The *ArrayOfWindows* parameter must specify either a zero- or one-dimensioned dynamic array or a single-dimensioned fixed array. If the array is dynamic, then it will be redimensioned to exactly hold the new number of elements. For fixed arrays, each array element is first erased, then the new elements are placed into the array. If there are fewer elements than will fit in the array, then the remaining elements are unused. A runtime error results if the array is too small to hold the new elements.

After calling this function, use the **LBound** and **UBound** functions to determine the new size of the array.

Example
```
'This example minimizes all top-level windows.
Sub Main()
   Dim a() As HWND
   WinList a
   For i = 1 To UBound(a)
      WinMinimize a(i)
   Next i
End Sub
```

See Also **WinFind** (function).

Platform(s) Windows, Win32.

WinMaximize (statement)

Syntax `WinMaximize [`*`window_name$ | window_object`*`]`

Description Maximizes the given window.

Comments The **WinMaximize** statement requires the following parameters:

Parameter	Description
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is maximized.

This command differs from the **AppMaximize** command in that this command operates on the current window rather than the current top-level window.

Example

```
'This example maximizes all top-level windows.
Sub Main()
   Dim a() As HWND
   WinList a
   For i = 1 To UBound(a)
      WinMaximize a(i)
   Next i
End Sub
```

See Also **WinMinimize** (statement); **WinRestore** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

WinMinimize (statement)

Syntax `WinMinimize [`*window_name$* `|` *window_object*`]`

Description Minimizes the given window.

Comments The **WinMinimize** statement requires the following parameters:

Parameter	Description
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is minimized.

This command differs from the **AppMinimize** command in that this command operates on the current window rather than the current top-level window.

Example See example for **WinList** (statement).

See Also **WinMaximize** (statement); **WinRestore** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

WinMove (statement)

Syntax `WinMove` *x*`,`*y* `[`*window_name$* `|` *window_object*`]`

Description Moves the given window to the given *x*,*y* position.

Comments The **WinMove** statement requires the following parameters:

Parameter	Description
x,y	**Integer** coordinates given in twips that specify the new location for the window.
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is moved.

This command differs from the **AppMove** command in that this command operates on the current window rather than the current top-level window. When moving child windows, remember that the *x* and *y* coordinates are relative to the client area of the parent window.

Example

```
'This example moves Program Manager to upper left corner of the
'screen.
WinMove 0,0,"Program Manager"
```

See Also **WinSize** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

WinRestore (statement)

Syntax `WinRestore [`*window_name$* `|` *window_object*`]`

Description Restores the specified window to its restore state.

Comments Restoring a minimized window restores that window to it screen position before it was minimized. Restoring a maximized window resizes the window to its size previous to maximizing.

The **WinRestore** statement requires the following parameters:

Parameter	Description
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find"
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is restored.

This command differs from the **AppRestore** command in that this command operates on the current window rather than the current top-level window.

Example

```
'This example minimizes all top-level windows except for Program
'Manager.
Sub Main()
   Dim a() As HWND
   WinList a
   For i = 0 To UBound(a)
     WinMinimize a(i)
   Next I
   WinRestore "Program Manager"
End Sub
```

See Also **WinMaximize** (statement); **WinMinimize** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

WinSize (statement)

Syntax `WinSize` *width*`,`*height* `[,`*window_name$* `|` *window_object*`]`

Description Resizes the given window to the specified width and height.

Comments The **WinSize** statement requires the following parameters:

Parameter	Description
width,height	**Integer** coordinates given in twips that specify the new size of the window.
window_name$	**String** containing the name that appears on the desired application's title bar. Optionally, a partial name can be used, such as "Word" for "Microsoft Word." A hierarchy of windows can be specified by separating each window name with a vertical bar (|), as in the following example: `WinActivate "Notepad|Find"` In this example, the top-level windows are searched for a window whose title contains the word "Notepad". If found, the windows owned by the top level window are searched for one whose title contains the string "Find".
window_object	**HWND** object specifying the exact window to activate. This can be used in place of the *window_name$* parameter to indicate a specific window to activate.

If *window_name$* and *window_object* are omitted, then the window with the focus is resized.

This command differs from the **AppSize** command in that this command operates on the current window rather than the current top-level window.

Example

```
'This example runs and resizes Notepad.
Sub Main()
   Dim NotepadApp As HWND
   id = Shell("Notepad.exe")
   set NotepadApp = WinFind("Notepad")
   WinSize 4400,8500,NotepadApp
End Sub
```

See Also **WinMove** (statement).

Platform(s) Windows, Win32.

Platform Notes **Windows, Win32:** On all Windows platforms, the current window can be an MDI child window, a pop-up window, or a top-level window.

Word$ (function)

Syntax `Word$(text$,first[,last])`

Description Returns a **String** containing a single word or sequence of words between *first* and *last*.

Comments The **Word$** function requires the following parameters:

Parameter	Description
text$	**String** from which the sequence of words will be extracted.
first	**Integer** specifying the index of the first word in the sequence to return. If *last* is not specified, then only that word is returned.
last	**Integer** specifying the index of the last word in the sequence to return. If *last* is specified, then all words between *first* and *last* will be returned, including all spaces, tabs, and end-of-lines that occur between those words.

Words are separated by any nonalphanumeric characters such as spaces, tabs, end-of-lines, and punctuation. On multi-byte and wide character platforms, double-byte spaces are treated as separators as well. Embedded null characters are treated as regular characters.

If *first* is greater than the number of words in *text$*, then a zero-length string is returned.

If *last* is greater than the number of words in *text$*, then all words from *first* to the end of the text are returned.

Example

```
'This example finds the name "Stuart" in a string and then
'extracts two words from the string.
Sub Main()
   s$ = "My last name is Williams; Stuart is my surname."
   c$ = Word$(s$,5,6)
   MsgBox "The extracted name is: " & c$
End Sub
```

See Also **Item$** (function); **ItemCount** (function); **Line$** (function); **LineCount** (function); **WordCount** (function).

Platform(s) All.

WordCount (function)

Syntax `WordCount(text$)`

Description Returns an **Integer** representing the number of words in the specified text.

Comments Words are separated by spaces, tabs, and end-of-lines. Embedded null characters are treated as regular characters.

Example

```
'This example counts the number of words in a particular string.
Sub Main()
   s$ = "My last name is Williams; Stuart is my surname."
   i% = WordCount(s$)
   MsgBox "'" & s$ & "' has " & i% & " words."
End Sub
```

See Also **Item$** (function); **ItemCount** (function); **Line$** (function); **LineCount** (function); **Word$** (function).

Platform(s) All.

Write# (statement)

Syntax `Write [#]`*filenumber* `[,`*expressionlist*`]`

Description Writes a list of expressions to a given sequential file.

Comments The file referenced by *filenumber* must be opened in either **Output** or **Append** mode.

The *filenumber* parameter is an **Integer** used by BasicScript to refer to the open file—the number passed to the **Open** statement.

The following summarizes how variables of different types are written:

Data Type	Description
Any numeric type	Written as text. There is no leading space, and the period is always used as the decimal separator.
String	Written as text, enclosed within quotes.
Empty	No data is written.
Null	Written as **#NULL#**.
Boolean	Written as **#TRUE#** or **#FALSE#**.
Date	Written using the universal date format: *#YYYY-MM-DD HH:MM:SS#*
User-defined errors	Written as **#ERROR** *ErrorNumber***#**, where *ErrorNumber* is the value of the user-defined error. The word **ERROR** is not translated.

The **Write** statement outputs variables separated with commas. After writing each expression in the list, **Write** outputs an end-of-line.

The **Write** statement can only be used with files opened in **Output** or **Append** mode.

Example

```
'This example opens a file for sequential write, then writes ten
'records into the file with the values 10...50. Then the file is
'closed and reopened for read, and the records are read with the
'Input statement. The results are displayed in a dialog box.
Sub Main()
   Open "test.dat" For Output Access Write As #1
   For x = 1 To 10
      r% = x * 10
      Write #1,x,r%
   Next x
   Close
   Open "test.dat" For Input Access Read As #1
   For x = 1 To 10
      Input #1,a%,b%
      message = message & "Record " & a% & ": " & b% & Basic.Eoln$
   Next x
   MsgBox message
   Close
End Sub
```

See Also **Open** (statement); **Put** (statement); **Print#** (statement).

Platform(s) All.

WriteIni (statement)

Syntax `WriteIni` *section$*`,`*ItemName$*`,`*value$*`[,`*filename$*`]`

Description Writes a new value into an ini file.

Comments The **WriteIni** statement requires the following parameters:

Parameter	Description
section$	**String** specifying the section that contains the desired variables, such as "Windows." Section names are specified without the enclosing brackets.
ItemName$	**String** specifying which item from within the given section you want to change. If *ItemName$* is a zero-length string (""), then the entire section specified by *section$* is deleted.
value$	**String** specifying the new value for the given item. If *value$* is a zero-length string (""), then the item specified by *ItemName$* is deleted from the ini file.
filename$	**String** specifying the name of the ini file.

Example

```
'This example sets the txt extension to be associated with
'Notepad.
```

```
Sub Main()
   WriteIni "Extensions","txt", _
      "c:\windows\notepad.exe ^.txt","win.ini"
End Sub
```

See Also **ReadIni$** (function); **ReadIniSection** (statement).

Platform(s) Windows, Win32, OS/2.

Platform Notes **Windows, Win32:** Under Windows and Win32, if *filename$* is not specified, the win.ini file is used.

If the *filename$* parameter does not include a path, then this statement looks for ini files in the Windows directory.

Xor (operator)

Syntax *result* = *expression1* Xor *expression2*

Description Performs a logical or binary exclusion on two expressions.

Comments If both expressions are either **Boolean**, **Boolean** variants, or **Null** variants, then a logical exclusion is performed as follows:

If *expression1* **is**	**and** *expression2* **is**	**then the** *result* **is**
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

If either expression is **Null**, then **Null** is returned.

Binary Exclusion

If the two expressions are **Integer**, then a binary exclusion is performed, returning an **Integer** result. All other numeric types (including **Empty** variants) are converted to **Long**, and a binary exclusion is then performed, returning a **Long** result.

Binary exclusion forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in *expression1* **is**	**and bit in** *expression2* **is**	**the** *result* **is**
1	1	0
0	1	1
1	0	1
0	0	0

Example

```
'This example builds a logic table for the XOR function and
'displays it.
Sub Main()
   For x = -1 To 0
      For y = -1 To 0
         z = x Xor y
         message = message & Format(x,"True/False") & " Xor "
         message = message & Format(y,"True/False") & " = "
         message = message & Format(z,"True/False") & Basic.Eoln$
      Next y
   Next x
   MsgBox message
End Sub
```

See Also Operator Precedence (topic); **Or** (operator); **Eqv** (operator); **Imp** (operator); **And** (operator).

Platform(s) All.

Year (function)

Syntax `Year(`*date*`)`

Description Returns the year of the date encoded in the specified date parameter. The value returned is between 100 and 9999 inclusive.

The *date* parameter is any expression representing a valid date.

Example

```
'This example returns the current year in a dialog box.
Sub Main()
   tdate$ = Date$
   tyear! = Year(DateValue(tdate$))
   MsgBox "The current year is: " & tyear$
End Sub
```

See Also **Day** (function); **Minute** (function); **Second** (function); **Month** (function); **Hour** (function); **Weekday** (function); **DatePart** (function).

Platform(s) All.

APPENDIX A

Language Elements by Platform

The following table lists all BasicScript language elements and specifies the platforms on which these language elements are supported.

BasicScript Language Elements by Platform

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
#Const	■	■	■	■	■	■	■
#If...Then...#Else	■	■	■	■	■	■	■
&	■	■	■	■	■	■	■
'	■	■	■	■	■	■	■
()	■	■	■	■	■	■	■
*	■	■	■	■	■	■	■
+	■	■	■	■	■	■	■
-	■	■	■	■	■	■	■
/	■	■	■	■	■	■	■
<	■	■	■	■	■	■	■
<=	■	■	■	■	■	■	■
<>	■	■	■	■	■	■	■
= (assignment)	■	■	■	■	■	■	■
= (operator)	■	■	■	■	■	■	■
>	■	■	■	■	■	■	■
>=	■	■	■	■	■	■	■
\	■	■	■	■	■	■	■
^	■	■	■	■	■	■	■
_	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Abs	■	■	■	■	■	■	■
ActivateControl	■	❒	❒	❒	❒	❒	❒
And	■	■	■	■	■	■	■
Any	■	■	■	■	■	■	■
AnswerBox	■	■	■	■	❒	■	❒
AppActivate	■	■	❒	■	❒	■	❒
AppClose	■	■	❒	■	❒	❒	❒
AppFileName$	■	❒	❒	■	❒	❒	❒
AppFind, AppFind$	■	■	❒	■	❒	❒	❒
AppGetActive$	■	■	❒	■	❒	❒	❒
AppGetPosition	■	■	❒	■	❒	❒	❒
AppGetState	■	■	❒	■	❒	❒	❒
AppHide	■	■	❒	■	❒	❒	❒
AppList	■	■	❒	■	❒	❒	❒
AppMaximize	■	■	❒	■	❒	❒	❒
AppMinimize	■	■	❒	■	❒	❒	❒
AppMove	■	■	❒	■	❒	❒	❒
AppRestore	■	■	❒	■	❒	❒	❒
AppSetState	■	■	❒	■	❒	❒	❒
AppShow	■	■	❒	■	❒	❒	❒
AppSize	■	■	❒	■	❒	❒	❒
AppType	■	■	❒	❒	❒	❒	❒
ArrayDims	■	■	■	■	■	■	■
ArraySort	■	■	■	■	■	■	■
Asc, AscB, AscW	■	■	■	■	■	■	■
AskBox, AskBox$	■	■	■	■	❒	■	❒
AskPassword, AskPassword$	■	■	■	■	❒	■	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Atn	■	■	■	■	■	■	■
Basic.Architecture	■	■	■	■	■	■	■
Basic.Capability	■	■	■	■	■	■	■
Basic.CodePage	■	■	■	■	■	■	■
Basic.Eoln$	■	■	■	■	■	■	■
Basic.FreeMemory	■	■	■	■	■	■	■
Basic.HomeDir$	■	■	■	■	■	■	■
Basic.Locale$	■	■	■	■	■	■	■
Basic.OperatingSystem$	■	■	■	■	■	■	■
Basic.OperatingSystemVendor$	■	■	■	■	■	■	■
Basic.OperatingSystemVersion$	■	■	■	■	■	■	■
Basic.OS	■	■	■	■	■	■	■
Basic.PathSeparator$	■	■	■	■	■	■	■
Basic.Processor$	■	■	■	■	■	■	■
Basic.ProcessorCount$	■	■	■	■	■	■	■
Basic.Version$	■	■	■	■	■	■	■
Beep	■	■	■	■	■	■	■
Begin Dialog	■	■	■	■	❒	■	❒
Boolean	■	■	■	■	■	■	■
ButtonEnabled	■	❒	❒	❒	❒	❒	❒
ButtonExists	■	❒	❒	❒	❒	❒	❒
Call	■	■	■	■	■	■	■
CancelButton	■	■	■	■	❒	■	❒
CBool	■	■	■	■	■	■	■
CCur	■	■	■	■	■	■	■
CDate, CVDate	■	■	■	■	■	■	■
CDbl	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
ChDir	■	■	■	■	■	■	■
ChDrive	■	■	❒	■	■	❒	❒
CheckBox	■	■	■	■	❒	■	❒
CheckBoxEnabled	■	❒	❒	❒	❒	❒	❒
CheckBoxExists	■	❒	❒	❒	❒	❒	❒
Choose	■	■	■	■	■	■	■
Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$	■	■	■	■	■	■	■
CInt	■	■	■	■	■	■	■
Clipboard$ (function)	■	■	❒	■	❒	■	❒
Clipboard$ (statement)	■	■	❒	■	❒	■	❒
Clipboard.Clear	■	■	❒	■	❒	■	❒
Clipboard.GetFormat	■	■	❒	■	❒	■	❒
Clipboard.GetText	■	■	❒	■	❒	■	❒
Clipboard.SetText	■	■	❒	■	❒	■	❒
CLng	■	■	■	■	■	■	■
Close	■	■	■	■	■	■	■
ComboBox	■	■	■	■	❒	■	❒
ComboBoxEnabled	■	❒	❒	❒	❒	❒	❒
ComboBoxExists	■	❒	❒	❒	❒	❒	❒
Command, Command$	■	■	■	■	■	■	■
Const	■	■	■	■	■	■	■
Cos	■	■	■	■	■	■	■
CreateObject	■	■	❒	❒	❒	■	❒
CSng	■	■	■	■	■	■	■
CStr	■	■	■	■	■	■	■
CurDir, CurDir$	■	■	■	■	■	■	■
Currency	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
CVar	■	■	■	■	■	■	■
CVErr	■	■	■	■	■	■	■
Date (data type)	■	■	■	■	■	■	■
Date, Date$ (functions)	■	■	■	■	■	■	■
Date, Date$ (statements)	■	■	■	■	■	■	■
DateAdd	■	■	■	■	■	■	■
DateDiff	■	■	■	■	■	■	■
DatePart	■	■	■	■	■	■	■
DateSerial	■	■	■	■	■	■	■
DateValue	■	■	■	■	■	■	■
Day	■	■	■	■	■	■	■
DDB	■	■	■	■	■	■	■
DDEExecute	■	■	❒	■	❒	❒	❒
DDEInitiate	■	■	❒	■	❒	❒	❒
DDEPoke	■	■	❒	■	❒	❒	❒
DDERequest, DDERequest$	■	■	❒	■	❒	❒	❒
DDESend	■	■	❒	■	❒	❒	❒
DDETerminate	■	■	❒	■	❒	❒	❒
DDETerminateAll	■	■	❒	■	❒	❒	❒
DDETimeOut	■	■	❒	■	❒	❒	❒
Declare	■	■	■	■	■	■	■
DefBool	■	■	■	■	■	■	■
DefCur	■	■	■	■	■	■	■
DefDate	■	■	■	■	■	■	■
DefDbl	■	■	■	■	■	■	■
DefInt	■	■	■	■	■	■	■
DefLng	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
DefObj	■	■	■	■	■	■	■
DefSng	■	■	■	■	■	■	■
DefStr	■	■	■	■	■	■	■
DefVar	■	■	■	■	■	■	■
DeleteSetting	■	■	❒	■	❒	❒	❒
Desktop.ArrangeIcons	■	❒	❒	❒	❒	❒	❒
Desktop.Cascade	■	❒	❒	❒	❒	❒	❒
Desktop.SetColors	■	❒	❒	❒	❒	❒	❒
Desktop.SetWallpaper	■	❒	❒	❒	❒	❒	❒
Desktop.Snapshot	■	❒	❒	❒	❒	❒	❒
Desktop.Tile	■	❒	❒	❒	❒	❒	❒
Dialog (function)	■	■	■	■	❒	■	❒
Dialog (statement)	■	■	■	■	❒	■	❒
Dim	■	■	■	■	■	■	■
Dir, Dir$	■	■	■	■	■	■	■
DiskDrives	■	■	❒	❒	■	❒	❒
DiskFree	■	■	❒	❒	■	❒	❒
DlgCaption	■	■	■	■	■	■	❒
DlgControlId	■	■	■	■	❒	■	❒
DlgEnable (function)	■	■	■	■	❒	■	❒
DlgEnable (statement)	■	■	■	■	❒	■	❒
DlgFocus (function)	■	■	■	■	❒	■	❒
DlgFocus (statement)	■	■	■	■	❒	■	❒
DlgListBoxArray (function)	■	■	■	■	❒	■	❒
DlgListBoxArray (statement)	■	■	■	■	❒	■	❒
DlgProc	■	■	■	■	❒	■	❒
DlgSetPicture	■	■	■	■	❒	■	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
DlgText (statement)	■	■	■	■	❒	■	❒
DlgText$ (function)	■	■	■	■	❒	■	❒
DlgValue (function)	■	■	■	■	❒	■	❒
DlgValue (statement)	■	■	■	■	❒	■	❒
DlgVisible (function)	■	■	■	■	❒	■	❒
DlgVisible (statement)	■	■	■	■	❒	■	❒
Do...Loop	■	■	■	■	■	■	■
DoEvents (function)	■	■	■	■	■	■	■
DoEvents (statement)	■	■	■	■	■	■	■
DoKeys	■	❒	❒	❒	❒	❒	❒
Double	■	■	■	■	■	■	■
DropListBox	■	■	■	■	❒	■	❒
EditEnabled	■	❒	❒	❒	❒	❒	❒
EditExists	■	❒	❒	❒	❒	❒	❒
End	■	■	■	■	■	■	■
Environ, Environ$	■	■	■	■	■	■	■
Eof	■	■	■	■	■	■	■
Eqv	■	■	■	■	■	■	■
Erase	■	■	■	■	■	■	■
Erl	■	■	■	■	■	■	■
Err.Clear	■	■	■	■	■	■	■
Err.Description	■	■	■	■	■	■	■
Err.HelpContext	■	■	■	■	■	■	■
Err.HelpFile	■	■	■	■	■	■	■
Err.LastDLLError	❒	■	❒	■	❒	❒	❒
Err.Number	■	■	■	■	■	■	■
Err.Raise	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Err.Source	■	■	■	■	■	■	■
Error	■	■	■	■	■	■	■
Error, Error$	■	■	■	■	■	■	■
Exit Do	■	■	■	■	■	■	■
Exit For	■	■	■	■	■	■	■
Exit Function	■	■	■	■	■	■	■
Exit Sub	■	■	■	■	■	■	■
Exp	■	■	■	■	■	■	■
FileAttr	■	■	■	■	■	■	■
FileCopy	■	■	■	■	■	■	■
FileDateTime	■	■	■	■	■	■	■
FileDirs	■	■	■	■	■	■	■
FileExists	■	■	■	■	■	■	■
FileLen	■	■	■	■	■	■	■
FileList	■	■	■	■	■	■	■
FileParse$	■	■	■	■	■	■	■
FileType	■	❒	❒	❒	❒	❒	❒
Fix	■	■	■	■	■	■	■
For...Each	■	■	■	■	■	■	■
For...Next	■	■	■	■	■	■	■
Format, Format$	■	■	■	■	■	■	■
FreeFile	■	■	■	■	■	■	■
Function...End Function	■	■	■	■	■	■	■
Fv	■	■	■	■	■	■	■
Get	■	■	■	■	■	■	■
GetAllSettings	■	■	❒	■	❒	❒	❒
GetAttr	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
GetCheckBox	■	❒	❒	❒	❒	❒	❒
GetComboBoxItem$	■	❒	❒	❒	❒	❒	❒
GetComboBoxItemCount	■	❒	❒	❒	❒	❒	❒
GetEditText$	■	❒	❒	❒	❒	❒	❒
GetListBoxItem$	■	❒	❒	❒	❒	❒	❒
GetListBoxItemCount	■	❒	❒	❒	❒	❒	❒
GetObject	■	■	❒	❒	❒	■	❒
GetOption	■	❒	❒	❒	❒	❒	❒
GetSetting	■	■	❒	■	❒	❒	❒
Global	■	■	■	■	■	■	■
GoSub	■	■	■	■	■	■	■
Goto	■	■	■	■	■	■	■
GroupBox	■	■	■	■	❒	■	❒
HelpButton	■	■	■	■	❒	■	❒
Hex, Hex$	■	■	■	■	■	■	■
HLine	■	■	❒	❒	❒	❒	❒
Hour	■	■	■	■	■	■	■
HPage	■	■	❒	❒	❒	❒	❒
HScroll	■	■	❒	❒	❒	❒	❒
HWND	■	■	❒	❒	❒	❒	❒
HWND.Value	■	■	❒	❒	❒	❒	❒
If...Then...Else	■	■	■	■	■	■	■
IIf	■	■	■	■	■	■	■
IMEStatus	■	■	■	■	❒	■	❒
Imp	■	■	■	■	■	■	■
Inline	■	■	■	■	■	■	■
Input#	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Input, Input$, InputB, InputB$	■	■	■	■	■	■	■
InputBox, InputBox$	■	■	■	■	❒	■	❒
InStr, InstrB	■	■	■	■	■	■	■
Int	■	■	■	■	■	■	■
Integer	■	■	■	■	■	■	■
IPmt	■	■	■	■	■	■	■
IRR	■	■	■	■	■	■	■
Is	■	■	■	■	■	■	■
IsDate	■	■	■	■	■	■	■
IsEmpty	■	■	■	■	■	■	■
IsError	■	■	■	■	■	■	■
IsMissing	■	■	■	■	■	■	■
IsNull	■	■	■	■	■	■	■
IsNumeric	■	■	■	■	■	■	■
IsObject	■	■	■	■	■	■	■
Item$	■	■	■	■	■	■	■
ItemCount	■	■	■	■	■	■	■
Kill	■	■	■	■	■	■	■
LBound	■	■	■	■	■	■	■
LCase, LCase$	■	■	■	■	■	■	■
Left, Left$, LeftB, LeftB$	■	■	■	■	■	■	■
Len, LenB	■	■	■	■	■	■	■
Let	■	■	■	■	■	■	■
Like	■	■	■	■	■	■	■
Line Input #	■	■	■	■	■	■	■
Line$	■	■	■	■	■	■	■
LineCount	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
ListBox	■	■	■	■	❒	■	❒
ListBoxEnabled	■	❒	❒	❒	❒	❒	❒
ListBoxExists	■	❒	❒	❒	❒	❒	❒
Loc	■	■	■	■	■	■	■
Lock	■	■	■	■	■	■	■
Lof	■	■	■	■	■	■	■
Log	■	■	■	■	■	■	■
Long	■	■	■	■	■	■	■
LSet	■	■	■	■	■	■	■
LTrim, LTrim$	■	■	■	■	■	■	■
MacID	❒	❒	❒	❒	❒	■	❒
MacScript	❒	❒	❒	❒	❒	■	❒
Main	■	■	■	■	■	■	■
Mci	■	❒	❒	❒	❒	❒	❒
Menu	■	❒	❒	❒	❒	❒	❒
MenuItemChecked	■	❒	❒	❒	❒	❒	❒
MenuItemEnabled	■	❒	❒	❒	❒	❒	❒
MenuItemExists	■	❒	❒	❒	❒	❒	❒
Mid, Mid$, MidB, MidB$ (functions)	■	■	■	■	■	■	■
Mid, Mid$, MidB, MidB$ (statements)	■	■	■	■	■	■	■
Minute	■	■	■	■	■	■	■
MIRR	■	■	■	■	■	■	■
MkDir	■	■	■	■	■	■	■
Mod	■	■	■	■	■	■	■
Month	■	■	■	■	■	■	■
Msg.Close	■	■	❒	❒	❒	❒	❒
Msg.Open	■	■	❒	❒	❒	❒	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Msg.Text	■	■	❒	❒	❒	❒	❒
Msg.Thermometer	■	■	❒	❒	❒	❒	❒
MsgBox (function)	■	■	■	■	❒	■	❒
MsgBox (statement)	■	■	■	■	❒	■	❒
Name	■	■	■	■	■	■	■
Net.AddCon$	■	■	❒	❒	❒	❒	❒
Net.Browse$	■	■	❒	❒	❒	❒	❒
Net.CancelCon	■	■	❒	❒	❒	❒	❒
Net.Dialog	■	❒	❒	❒	❒	❒	❒
Net.GetCaps	■	■	❒	❒	❒	❒	❒
Net.GetCon$	■	■	❒	❒	❒	❒	❒
Net.User$	■	■	❒	❒	❒	❒	❒
Not	■	■	■	■	■	■	■
Now	■	■	■	■	■	■	■
NPer	■	■	■	■	■	■	■
Npv	■	■	■	■	■	■	■
Object	■	■	❒	❒	❒	■	❒
Oct, Oct$	■	■	■	■	■	■	■
OKButton	■	■	■	■	❒	■	❒
On Error	■	■	■	■	■	■	■
Open	■	■	■	■	■	■	■
OpenFilename$	■	■	■	■	❒	■	❒
Option Base	■	■	■	■	■	■	■
Option Compare	■	■	■	■	■	■	■
Option CStrings	■	■	■	■	■	■	■
Option Default	■	■	■	■	■	■	■
Option Explicit	■	■	■	■	■	■	■

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
OptionButton	■	■	■	■	❒	■	❒
OptionEnabled	■	❒	❒	❒	❒	❒	❒
OptionExists	■	❒	❒	❒	❒	❒	❒
OptionGroup	■	■	■	■	❒	■	❒
Or	■	■	■	■	■	■	■
Picture	■	■	■	■	❒	■	❒
PictureButton	■	■	■	■	❒	■	❒
Pmt	■	■	■	■	■	■	■
PopupMenu	■	■	❒	❒	❒	❒	❒
PPmt	■	■	■	■	■	■	■
Print	■	■	■	■	■	■	■
Print #	■	■	■	■	■	■	■
PrinterGetOrientation	■	❒	❒	❒	❒	❒	❒
PrinterSetOrientation	■	❒	❒	❒	❒	❒	❒
PrintFile	■	❒	❒	❒	❒	❒	❒
Private	■	■	■	■	■	■	■
Public	■	■	■	■	■	■	■
PushButton	■	■	■	■	❒	■	❒
Put	■	■	■	■	■	■	■
Pv	■	■	■	■	■	■	■
QueEmpty	■	❒	❒	❒	❒	❒	❒
QueFlush	■	❒	❒	❒	❒	❒	❒
QueKeyDn	■	❒	❒	❒	❒	❒	❒
QueKeys	■	❒	❒	❒	❒	❒	❒
QueKeyUp	■	❒	❒	❒	❒	❒	❒
QueMouseClick	■	❒	❒	❒	❒	❒	❒
QueMouseDblClk	■	❒	❒	❒	❒	❒	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
QueMouseDblDn	■	❒	❒	❒	❒	❒	❒
QueMouseDn	■	❒	❒	❒	❒	❒	❒
QueMouseMove	■	❒	❒	❒	❒	❒	❒
QueMouseMoveBatch	■	❒	❒	❒	❒	❒	❒
QueMouseUp	■	❒	❒	❒	❒	❒	❒
QueSetRelativeWindow	■	❒	❒	❒	❒	❒	❒
Random	■	■	■	■	■	■	■
Randomize	■	■	■	■	■	■	■
Rate	■	■	■	■	■	■	■
ReadINI$	■	■	❒	■	❒	❒	❒
ReadINISection	■	■	❒	■	❒	❒	❒
ReDim	■	■	■	■	■	■	■
REM	■	■	■	■	■	■	■
Reset	■	■	■	■	■	■	■
Resume	■	■	■	■	■	■	■
Return	■	■	■	■	■	■	■
Right, Right$, RightB, RightB$	■	■	■	■	■	■	■
RmDir	■	■	■	■	■	■	■
Rnd	■	■	■	■	■	■	■
RSet	■	■	■	■	■	■	■
RTrim, RTrim$	■	■	■	■	■	■	■
SaveFileName$	■	■	■	■	❒	■	❒
SaveSetting	■	■	❒	■	❒	❒	❒
Screen.DlgBaseUnitsX	■	■	❒	❒	❒	❒	❒
Screen.DlgBaseUnitsY	■	■	❒	❒	❒	❒	❒
Screen.Height	■	■	❒	❒	❒	❒	❒
Screen.TwipsPerPixelX	■	■	❒	❒	❒	❒	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
Screen.TwipsPerPixelY	■	■	❒	❒	❒	❒	❒
Screen.Width	■	■	❒	❒	❒	❒	❒
Second	■	■	■	■	■	■	■
Seek (function)	■	■	■	■	■	■	■
Seek (statement)	■	■	■	■	■	■	■
Select...Case	■	■	■	■	■	■	■
SelectBox	■	■	■	■	❒	■	❒
SelectButton	■	❒	❒	❒	❒	❒	❒
SelectComboboxItem	■	❒	❒	❒	❒	❒	❒
SelectListboxItem	■	❒	❒	❒	❒	❒	❒
SendKeys	■	■	❒	❒	❒	❒	❒
Set	■	■	■	■	■	■	■
SetAttr	■	■	■	■	■	■	■
SetCheckbox	■	❒	❒	❒	❒	❒	❒
SetEditText	■	❒	❒	❒	❒	❒	❒
SetOption	■	❒	❒	❒	❒	❒	❒
Sgn	■	■	■	■	■	■	■
Shell	■	■	■	■	■	■	■
Sin	■	■	■	■	■	■	■
Single	■	■	■	■	■	■	■
Sleep	■	■	■	■	■	■	■
Sln	■	■	■	■	■	■	■
Space, Space$	■	■	■	■	■	■	■
Spc	■	■	■	■	■	■	■
SQLBind	■	■	❒	❒	❒	❒	❒
SQLClose	■	■	❒	❒	❒	❒	❒
SQLError	■	■	❒	❒	❒	❒	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
SQLExecQuery	■	■	❒	❒	❒	❒	❒
SQLGetSchema	■	■	❒	❒	❒	❒	❒
SQLOpen	■	■	❒	❒	❒	❒	❒
SQLRequest	■	■	❒	❒	❒	❒	❒
SQLRetrieve	■	■	❒	❒	❒	❒	❒
SQLRetrieveToFile	■	■	❒	❒	❒	❒	❒
Sqr	■	■	■	■	■	■	■
Stop	■	■	■	■	■	■	■
Str, Str$	■	■	■	■	■	■	■
StrComp	■	■	■	■	■	■	■
StrConv	■	■	■	■	■	■	■
String	■	■	■	■	■	■	■
String, String$	■	■	■	■	■	■	■
Sub...End Sub	■	■	■	■	■	■	■
Switch	■	■	■	■	■	■	■
SYD	■	■	■	■	■	■	■
System.Exit	■	❒	❒	❒	❒	❒	❒
System.FreeMemory	■	■	❒	❒	❒	❒	❒
System.FreeResources	■	❒	❒	❒	❒	❒	❒
System.MouseTrails	■	■	❒	❒	❒	❒	❒
System.Restart	■	■	❒	❒	❒	❒	❒
System.TotalMemory	■	■	❒	❒	❒	❒	❒
System.WindowsDirectory$	■	■	❒	❒	❒	❒	❒
System.WindowsVersion$	■	■	❒	❒	❒	❒	❒
Tab	■	■	■	■	■	■	■
Tan	■	■	■	■	■	■	■
Text	■	■	■	■	❒	■	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
TextBox	■	■	■	■	❒	■	❒
Time, Time$ (functions)	■	■	■	■	■	■	■
Time, Time$ (statements)	■	■	■	■	■	■	■
Timer	■	■	■	■	■	■	■
TimeSerial	■	■	■	■	■	■	■
TimeValue	■	■	■	■	■	■	■
Trim, Trim$	■	■	■	■	■	■	■
Type	■	■	■	■	■	■	■
TypeName	■	■	■	■	■	■	■
TypeOf	■	■	■	■	■	■	■
UBound	■	■	■	■	■	■	■
UCase, UCase$	■	■	■	■	■	■	■
UnLock	■	■	■	■	■	■	■
Val	■	■	■	■	■	■	■
Variant	■	■	■	■	■	■	■
VarType	■	■	■	■	■	■	■
Viewport.Clear	■	❒	❒	❒	❒	❒	❒
Viewport.Close	■	❒	❒	❒	❒	❒	❒
Viewport.Open	■	❒	❒	❒	❒	❒	❒
VLine	■	■	❒	❒	❒	❒	❒
VPage	■	■	❒	❒	❒	❒	❒
VScroll	■	■	❒	❒	❒	❒	❒
Weekday	■	■	■	■	■	■	■
While...Wend	■	■	■	■	■	■	■
Width#	■	■	■	■	■	■	■
WinActivate	■	■	❒	❒	❒	❒	❒
WinClose	■	■	❒	❒	❒	❒	❒

BasicScript Language Elements by Platform (Continued)

Language Element	Win	Win32	UNIX	OS/2	NetWare	Macintosh	OpenVMS
WinFind	■	■	❒	❒	❒	❒	❒
WinList	■	■	❒	❒	❒	❒	❒
WinMaximize	■	■	❒	❒	❒	❒	❒
WinMinimize	■	■	❒	❒	❒	❒	❒
WinMove	■	■	❒	❒	❒	❒	❒
WinRestore	■	■	❒	❒	❒	❒	❒
WinSize	■	■	❒	❒	❒	❒	❒
Word$	■	■	■	■	■	■	■
WordCount	■	■	■	■	■	■	■
Write #	■	■	■	■	■	■	■
WriteIni	■	■	❒	■	❒	❒	❒
Xor	■	■	■	■	■	■	■
Year	■	■	■	■	■	■	■

APPENDIX B

Runtime Error Messages

This section contains lists of all the error messages that BasicScript may display at runtime. It is divided into two subsections, the first describing errors messages compatible with “standard” Basic as implemented by Microsoft Visual Basic and the second describing error messages specific to BasicScript.

A few error messages contain placeholders, which get replaced by the runtime when forming the completed runtime error message. These placeholders appear in the following list as the italicized word *placeholder*.

Visual Basic–Compatible Error Messages

Error Number	Error Message
3	Return without GoSub
5	Invalid procedure call
6	Overflow
7	Out of memory
9	Subscript out of range
10	This array is fixed or temporarily locked
11	Division by zero
13	Type mismatch
14	Out of string space
18	User interrupt occurred
20	Resume without error
26	Dialog needs End Dialog or push button
28	Out of stack space
35	Sub or Function not defined
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
61	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

Error Number	Error Message
93	Invalid pattern string
94	Invalid use of Null
139	Only one user dialog may be up at any time
140	Dialog control identifier does not match any current control
141	The *placeholder* statement is not available on this dialog control type
143	The dialog control with the focus may not be disabled or hidden
144	Focus may not be set to a hidden or disabled control
150	Dialog control identifier is already defined
163	This statement can only be used when a user dialog is active
260	No timer available
281	No more DDE channels
282	No foreign application responded to a DDE initiate
283	Multiple applications responded to a DDE initiate
285	Foreign application won't perform DDE method or operation
286	Timeout while waiting for DDE response
287	User pressed Escape key during DDE operation
288	Destination is busy
289	Data not provided in DDE operation
290	Data in wrong format
291	Foreign application quit
292	DDE conversation closed or changed
295	Message queue filled; DDE message lost
298	DDE requires ddeml.dll
380	Invalid property value
423	Property or method not found
424	Object required
429	OLE Automation server can't create object
430	Class doesn't support OLE Automation
431	OLE Automation server cannot load file
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
442	Connection to type library or object library for remote process has been lost. Press OK for dialog to remove reference.
443	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 locale setting

Error Number	Error Message
448	Named argument not found
449	Argument not optional
450	Wrong number of arguments or invalid property assignment
451	Object not a collection
452	Invalid ordinal
453	Specified DLL function not found
454	Code resource not found
455	Code resource lock error
460	Invalid Clipboard format
481	Invalid picture
520	Can’t empty clipboard
521	Can’t open clipboard
600	Set value not allowed on collections
601	Get value not allowed on collections
603	ODBC - SQLAllocEnv failure
604	ODBC - SQLAllocConnect failure
608	ODBC - SQLFreeConnect error
610	ODBC - SQLAllocStmt failure
3129	Invalid SQL statement; expected 'DELETE', 'INSERT', 'PROCEDURE', 'SELECT', or 'UPDATE'
3146	ODBC -- call failed.
3148	ODBC -- connection failed.
3276	Invalid database ID

BasicScript-Specific Error Messages

Error Number	Error Message
800	Incorrect Windows version
801	Too many dimensions
802	Can’t find window
803	Can’t find menu item
804	Another queue is being flushed
805	Can’t find control
806	Bad channel number
807	Requested data not available

Error Number	Error Message
808	Can’t create popup menu
810	Command failed
811	Network error
812	Network function not supported
813	Bad password
814	Network access denied
815	Network function busy
816	Queue overflow
817	Too many dialog controls
818	Can’t find listbox/combobox item
819	Control is disabled
820	Window is disabled
821	Can’t write to INI file
822	Can’t read from INI file
823	Can’t copy file onto itself
824	OLE Automation unknown object name
825	Redimension of a fixed array
826	Can’t load and initialize extension
827	Can’t find extension
828	Unsupported function or statement
829	Can’t find ODBC libraries
830	OLE Automation Lbound or Ubound on non-Array value
831	Incorrect definition for dialog procedure
832	Incorrect number of arguments for intermodule call
833	OLE Automation object does not exist
834	Access to OLE Automation object denied
835	OLE initialization error
836	OLE Automation method returned unsupported type
837	OLE Automation method did not return a value
838	OLE automation error - the remote procedure call connection terminated
839	OLE automation error - the RPC server is unavailable

Error Number	Error Message
840	OLE automation error - the RPC server is too busy to complete this operation
841	OLE automation error - the remote procedure call failed
842	OLE automation error - the remove procedure call failed and did not execute

APPENDIX C

Compiler Error Messages

The following table contains a list of all the errors that may be generated by the BasicScript compiler. With some errors, the compiler changes placeholders within the error to text from the script being compiled. These placeholders are represented in this table by the italicized word *placeholder.*

Error Number	Error Message
1	Variable Required - Can't assign to this expression
2	Letter range must be in ascending order
3	Redefinition of default type
4	Out of storage for variables
5	Type-declaration character doesn't match defined type
6	Expression too complex
7	Cannot assign whole array
8	Assignment variable and expression are different types
9	Type-declaration character not allowed for function with explicit type
10	Array type mismatch in parameter
11	Array type expected for parameter
12	Array type unexpected for parameter
13	Integer expression expected for an array index
14	Integer expression expected
15	String expression expected
16	Identifier is already a user defined type
17	Property value is the incorrect type
18	Left of "." must be an object, structure, or dialog
19	Invalid string operator
20	Can't apply operator to array type
21	Operator type mismatch
22	"*placeholder*" is not a variable
23	"*placeholder*" is not an array variable or a function
24	Unknown *placeholder* "*placeholder*"
25	Out of memory
26	*placeholder*: Too many parameters encountered

Error Number	Error Message
27	*placeholder*: Missing parameter(s)
28	*placeholder*: Type mismatch in parameter *placeholder*
29	Missing label "*placeholder*"
30	Too many nested statements
31	Encountered new-line in string
32	Overflow in decimal value
33	Overflow in hex value
34	Overflow in octal value
35	Expression is not constant
36	Not inside a Do statement
37	Type-declaration character not allowed for parameter with explicit type
39	Can't pass an array by value
40	"*placeholder*" is already declared as a parameter
41	Variable name used as label name
42	Duplicate label
43	Not inside a function
44	Not inside a sub
46	Can't assign to function
47	Identifier is already a variable
48	Unknown type
49	Variable is not an array type
50	Can't redimension an array to a different type
51	Identifier is not a string array variable
52	0 expected
54	*placeholder* is not an assignable property of the object
56	*placeholder* is not a method of the object
57	*placeholder* is not a property of the object
58	Expecting 0 or 1
59	Boolean expression expected
60	Numeric expression expected
61	Numeric type For variable expected
62	For...Next variable mismatch
63	Out of string storage space
64	Out of identifier storage space
68	Division by zero
69	Overflow in expression
70	Floating-point expression expected
72	Invalid floating-point operator

Error Number	Error Message
74	Single character expected
75	Subroutine identifier can't have a type-declaration character
76	Script is too large to be compiled
77	Variable type expected
78	Types and dialog variables can’t be passed by value
79	Can't assign to user or dialog type variable
80	Maximum string length exceeded
81	Identifier name already in use as a variable
84	Operator cannot be used on an object
85	*placeholder* is not a property or method of the object
86	Label cannot contain type-declaration character
87	Type-declaration character mismatch in *placeholder*
88	Destination name is already a constant
89	Can't assign to constant
91	Identifier too long
92	Expecting string or structure expression
93	Can't assign to expression
94	Dialog and Object types are not supported in this context
95	Array expression not supported as parameter
96	Dialogs, objects, and structure expressions are not supported as a parameter
97	Invalid numeric operator
98	Invalid structure element name following "."
99	Access value can't be used with specified mode
101	Invalid operator for object
102	Can't LSet a type with a variable-length string
103	Syntax error
105	No members defined
106	Duplicate type member
107	Set is for object assignments
109	Invalid character in octal number
110	Invalid numeric prefix: expecting &H or &O
111	End of script encountered in comment: expecting */
112	Misplaced line continuation
113	Invalid escape sequence
114	Missing End Inline
115	Statement expected
116	ByRef argument mismatch

Error Number	Error Message
117	Integer overflow
118	Long overflow
119	Single overflow
120	Double overflow
121	Currency overflow
122	Optional argument must be Variant
123	Parameter must be optional
124	Parameter is not optional
125	Expected: Lib
126	Illegal external function return type
127	Illegal function return type
128	Variable not defined
129	No default property for the object
130	The object does not have an assignable default property
131	Parameters cannot be fixed length strings
132	Invalid length for a fixed length string
133	Return type is different from a prior declaration
134	Private variable too large. Storage space exceeded
135	Public variables too large. Storage space exceeded
136	Type-declaration character not allowed for variable with explicit type
137	Missing parameters are not allowed when using named parameters
138	An unnamed parameter was found following a named parameter
139	Unknown parameter name: *placeholder*
140	Duplicate parameter name: *placeholder*
141	Expecting: #If, #ElseIf, #Else, #End If, or #Const
142	Invalid preprocessor directive
143	Expecting preprocessor variable
144	Expecting: =
145	Expecting: [end of line]
146	Expecting: <expression>
148	Expecting:)
149	Unexpected value
150	Expecting: #End If
151	Expecting: Then
152	Missing #End If
153	#Else encountered without #If
154	#ElseIf encountered without #If
155	#End If encountered without #If

Error Number	Error Message
156	Invalid use of Null
157	Type mismatch
158	Not a number
159	Duplicate subroutine definition
160	Duplicate function definition
161	MBCS characters not allowed in identifiers
162	Out of range
163	Invalid date
164	Date overflow
165	Expecting: <identifier>
166	Constant type and expression are different types
167	Invalid use of New
168	Encountered: *placeholder* Expecting: *placeholder*
169	For Each control variable on arrays must be a variant
170	For Each control variable on collections must be a variant or an object
171	For Each may not be used on an array of user-defined types or fixed-length strings
172	For Each may only iterate over an object collection or an array
173	Not inside a For...Next statement
174	Invalid use of parenthesis with property
175	Object does not support For Each
176	Improper use of method that does not return a value
177	Improper use of method that returns a value
178	Sub or Function not allowed
179	Overflow in binary value
180	Private statement not allowed

APPENDIX D

BasicScript Limitations

The following list contains important BasicScript limitations:

- Line numbers are not supported. Labels can be used in place of line numbers as targets for the Goto statement.
- Variable-length strings are limited in size to 65,528 bytes. This includes local, public, and private variable-length strings, as well as variable-length strings contained in structures and arrays.

  This byte limitation translates to 32,764 characters on Win32 platforms where each character requires 2-bytes of storage (BasicScript uses UNICODE for its internal string format on Win32 platforms). On multi-byte character platforms where variable-length strings can contain both 1 and 2 byte characters, the character limit depends on the number of 2-byte characters in the string. On single-byte character platforms, the character limit is the same as the byte limit.

  When appearing within structures and arrays, variable-length strings only require 2 bytes of storage, as their content is contained in a different data area called *string space*.
- The initial size of the string space is 8K, which expands automatically up to a maximum as determined by the application hosting BasicScript. Unless otherwise changed by the hosting application, the maximum size of the string space is 64K.

  String space contains all variable-length strings and arrays regardless of their scope.

  Note: The application hosting BasicScript may increase or decrease the maximum size of string space. Even so, under Windows 3.1, the maximum size of string space cannot exceed 1 MB regardless of the size set by the hosting application.
- The default stack size for executing scripts is 2,048 bytes. This space contains all local variables and passed parameters (arrays and variable-length strings only require 2-bytes of stack, as their contents are contained in string space).

  The stack is also used by the runtime for storage of intermediate values, so the actual stack space available for storage of local variables may be slightly less.

  Calls made to subroutines or functions in other scripts use the stack of the caller.

Note: The application hosting BasicScript may increase the size of the stack up to a maximum of 8K.

- The data area that holds each script's private variables is limited to 16K per script. This data space contains all private variables defined within the script (variable-length strings and arrays require only 2 bytes of storage in the private variable space, as their contents are stored in the string space).
- The data area that holds public variables is limited to 16K. This data space contains all public variables defined by all scripts (variable-length strings and arrays require only 2 bytes of storage in the public variable space, as their contents are stored in the string space).
- Fixed-length strings have the same maximum size as variable-length strings, but have a practical limit which is imposed by the data area from which they are allocated.

  Local fixed-length strings: If the maximum size of the stack is 2,048 bytes, then the largest local fixed-length string will be slightly less than 2,048 bytes. On Win32 platforms, since each character is 2 bytes, this translates to slightly less than 1,024 characters.

  Private and Public fixed-length strings: Since the maximum size of the storage for private and public variables is 16K, this means that the largest fixed-length string stored in either of these data areas is 16,384 characters. On Win32 platforms, this translates to 8192 characters, since each character is 2 bytes. Considering that there is likely to be other variables contained in these data areas, the actual limit may be much less.

  Fixed-length strings contained in arrays and structures are stored along with the other members of these compound data items, and are thus restricted in size to the limits from which their containing data items are allocated.
- The Visual Basic declaration modifiers Static and Shared are not supported.
- The size of a source script is limited to 65,534 characters under Windows 3.1. This limitation can be avoided by breaking up large scripts into smaller ones.

  On all other platforms, script size is limited by available memory.
- The maximum number of lines in a script is limited to 65,535 lines.
- A compiled script consists of p-code, constant initialized data, and symbolic information. On all platforms, the maximum size of the constant data is limited to 65,535 bytes. Similarly, the maximum size of the symbolic information is 65,535 bytes. (These limitations are rarely encountered, if ever.)

  Under Windows, the maximum size of the code is 65,535 bytes. On all other platforms, the maximum size of the code is limited only by available memory.

The 64K limitations under Windows can be avoided by breaking up large scripts into smaller ones, which is rarely necessary.

- Arrays can have up to 60 dimensions.
- Variable names are limited to 80 characters.
- Labels are limited to 80 characters.
- Each executing script contains a table of structures that track calls made to external routines. Each structure is approximately 88 bytes with an overall size limited to 64K.
- The number of open DDE channels is not fixed; rather, it is limited only by available memory and system resources.
- The number of open files is limited to 512 or the operating system limit, whichever is less.
- The maximum size of a string literal (a string enclosed within quotation marks) is limited to 1,024 bytes. (Strings can be concatenated using the concatenation [&] operator with the normal string limit of 32,764 bytes.)

  On wide-character systems (i.e., UNICODE on Win32 platforms), 1024 bytes ranslates to 512 characters. On single-byte, this translates to 1,024 characters. On multibyte systems, the maximum length depends on the number of 2-byte characters.
- The number of nesting levels (i.e., loops within loops) is limited by compiler memory.
- Queue playback buffer size is limited to 64K. With 10 bytes per event, this allows for 6,553 events.
- Each GoSub requires 4 bytes of the BasicScript runtime stack.
- Arrays and user-defined types cannot be passed to a method of an OLE Automation object.
- Arrays and user-defined types cannot be set as the value of a property of an OLE Automation object.
- Arrays and user-defined types cannot be returned from a method or property of an OLE Automation object.
- Array indexes must be in the following range:

  -32,768 <= *array-index* <=32,767
- The size of an array cannot exceed 32K. For example, an array of integers, each of which requires 2 bytes of storage, is limited to the following maximum number of elements:

  max_num_elements = (32,767 - *overhead*) / 2

where overhead is currently approximately 16 bytes.

- A maximum of 128 fonts can be used within a single user dialog, although the practical limitation imposed by the operating system may be less.

APPENDIX E

BasicScript/Visual Basic Differences

This appendix describes the differences between Visual Basic 4.0 and BasicScript version 2.2.

The following topics are covered:

- Arrays
- Constants
- Data Types
- Debugging
- Declarations
- Declare Statement
- Error Handling
- Floating-Point Numbers
- Currency Numbers
- Language Element Differences
- Natural Language Support
- Objects
- Parameter Passing
- Strings
- Variants
- Stack Size
- Expression Evaluation
- File Searching

Arrays

Visual Basic supports huge arrays, BasicScript does not.

BasicScript and Visual Basic differ in the way that elements are stored in memory. Visual Basic stores elements in column-major order such as FORTRAN, meaning that the leftmost dimension changes first. For example, consider the following statement:

```
Dim a(1 To 3,1 To 2)
```

In Visual Basic, the elements are stored in memory as follows:

```
a(1,1)
a(2,1)
a(3,1)
a(1,2)
a(2,2)
a(3,2)
```

BasicScript uses the same element ordering as C where the lower dimension changes first, as shown below:

```
a(1,1)
a(1,2)
a(2,1)
a(2,2)
a(3,1)
a(3,2)
```

This difference impacts code that passes arrays to external routines using Declare and the use of the **For...Each** statement.

Constants

Visual Basic supports shared constants (using the Public keyword). In BasicScript, constants must be repeated within each script in which they are used.

Visual Basic does not allow the concatenation of constant elements. For example, the following script compiles in BasicScript but not in Visual Basic:

```
Const t$ = "Hello" & Chr$(9) & "there."

Sub Main()
    MsgBox t$
End Sub
```

Visual Basic allows a user to redefine global constants at the subroutine/function level without affecting their global values; BasicScript does not. For example, the following script will compile and execute in Visual Basic but not in BasicScript:

```
Const t$ = "Hello"
```

```
Sub Main()
    Const t$ = "Good-bye"
    MsgBox t$
End Sub
```

Declarations

Visual Basic supports the Static keyword as a modifier for the Sub and Function statements. BasicScript supports use of this keyword with these statements syntactically, but has no effect symantecally.

A variable used in a comparison expression that hasn't been declared will be implicitly declared in Visual Basic. In BasicScript, this will be seen as an unresolved function:

```
Sub Main
    If a$ = "Hello" Then Beep
End Sub
```

In BasicScript, the above script will compile, but it gives a Sub or Function not defined error when executed. In Visual Basic, this will automatically declare a variable called a$ as a String.

Debugging

While debugging, the trace function will execute a single-line **If...Then** statement as multiple units, requiring two presses of the F8 key. The first trace will execute the condition and the second will execute one of the statements.

Declare Statements

Visual Basic supports shared Declare statements (using the Public keyword). In BasicScript, these must be declared in every script in which they are used.

BasicScript supports a superset of that functionality available in Visual Basic—namely, the additional calling conventions.

BasicScript and Visual Basic pass values to external routines in the same manner, with the following exceptions:

- BasicScript passes True or False as Boolean values (signed short in C). Visual Basic passes these as Boolean variants.
- Arrays are passed to external routines as OLE safearrays. BasicScript passes arrays as a pointer to the first array element.

- Variants are passed as internal variant structures in both BasicScript and Visual Basic. For all numeric values, the types are the same.

  The variant structure in both systems is a 4-byte type (a 32-bit integer—the same value as returned by the VarType function), followed by 4 bytes of slop, followed by the value of the variable, as shown below:

Bytes 0–3	Bytes 4–7	Bytes 8–15
VarType	Alignment slop	Value

Error Handling

The On Error Resume Next statement causes execution to continue on the next line rather than at the next statement. This difference is only visible when you have placed more than one statement on the same line, separated with a colon. For example, the following code displays nothing in BasicScript, while, in Visual Basic, will display a dialog:

```
Sub Main
    On Error Resume Next
    Error 10 : MsgBox "Hello, world."
End Sub
```

Floating-Point Numbers

In Visual Basic, floating-point numbers are interpreted as doubles unless they are explicitly accompanied by a type-declaration character. Thus, the following line assigns a Double in Visual Basic, whereas in BasicScript, it assigns a Single:

```
a = 0.00001
```

In BasicScript, additional checking is performed to determine whether a floating-point number can be accurately represented as a Single. If so, then the number is stored as a Single, requiring 4 bytes rather than 8.

The implications of this difference can be seen in the following code:

```
Dim a As Variant,b As Variant

a = 1000
b = .00001
a = a + b

MsgBox a
```

In Visual Basic, since the variables a and b are assigned Double values, the addition is performed between two doubles, resulting in the value 1000.00001. In BasicScript, on the other hand, a and b are assigned Single values, resulting in an addition between two singles. When these two singles are added, there is a loss of precision resulting in the value 1000.

In situations such as this, you should explicitly force the types using type-declaration characters. The above code can be rewritten as follows:

```
Dim a As Variant,b As Variant

a = 1000#
b = .00001#
a = a + b

MsgBox a              'BasicScript displays 1000.00001.
```

Currency Numbers

In Visual Basic, Double numbers do not convert to Currency numbers the same way. In Visual Basic, for example, the following script will fail:

```
Sub Main
    result = CCur("-1.401298E-45")
End Sub
```

The above fails in Visual Basic because the number being converted is known to be a Double. In BasicScript, any number between the valid range supported by Currency is convertible to Currency, even if the number is expressed in scientific notation or is extremely small (approaching zero).

Language Element Differences

Visual Basic and BasicScript use a slightly different syntax for the following SQL functions (due to BasicScript's lack of support for variant arrays):

```
SQLError
SQLGetSchema
SQLRetrieve
SQLRequest
```

In Visual Basic, the **GetAllSettings** function returns a variant containing an array. BasicScript does not support arrays within variants, and therefore takes an array variable as its last parameter.

The Visual Basic **Write** statement accepts commas, semi-colons, and spaces as parameter separators, much like the **Print** statement. In BasicScript, the Write statement

cannot accept semi-colons as space separators, nor will it accept trailing commas or semi-colons. Both the **Print** and **Write** statements in BasicScript reject spaces as parameters separators.

The **Const** statement in BasicScript can only be used outside the scope of any subroutine or function declaration. In Visual Basic, **Const** statements appearing within the definition of a subroutine or function have scope local to that routine.

BasicScript does not support any of the following Visual Basic language elements:

Language Element	Type
Array	Function
Exit Property	Statement
For Each...Next	Statement
LoadPicture	Function
On...Gosub	Statement
On...Goto	Statement
Option Private	Statement
Property Get...End Property	Statement
Property Let...End Property	Statement
Property Set...End Property	Statement
SavePicture	Statement
Screen.MousePointer	Property
Static	Statement
With...End With	Statement

Objects

BasicScript does not support any of Visual Basic's objects (except **Clipboard**, **Screen**, and a few others).

Strings

In BasicScript, variable-length strings within structures require 2 bytes of storage. In Visual Basic, variable-length strings within structures require 4 bytes of storage.

The implications of this difference can be seen in the following code:

```
Type Sample
    LastName As String
End Type

Sub Main
    Dim a As Sample
    MsgBox Len(a)
End Sub
```

In the above code, Visual Basic displays 4, whereas BasicScript displays 2.

In BasicScript, variable-length strings are limited to 32K in length. In Visual Basic, variable-length strings have no limits on their lengths.

Visual Basic will not accept strings in some functions expecting numbers such as Int and Fix. BasicScript allows strings as long as they are convertible to numbers.

```
Dim A As Variant
Abs(19)          'OK.
A = "10"
Abs(A)           'OK.
Abs("10")        'Works in BasicScript, not in Visual Basic
```

In BasicScript, these functions will accept any data type convertible to a number. If the data type is a **String**, BasicScript converts it to a Double.

Fixed-length strings within structures are size-adjusted upward to an even size. Thus, structures in BasicScript are always even-sized. Visual Basic allows fixed-length strings within structures to maintain an odd size.

Variants

Passing variants either by value or by reference to external routines (using the Declare statement) passes either the entire variant structure (ByVal) or a pointer to a variant structure (ByRef) used internally by BasicScript. This means that passing variants to an externally declared routine can only be done if that routine is aware of the internal variant structure used by BasicScript.

Visual Basic supports variant arrays; BasicScript does not. This includes use of the Array function.

Passing Variants by Reference

In Visual Basic, variants cannot be passed by reference to user-defined routines accepting nonvariant parameters. For example, the following will not work in Visual Basic:

```
Sub Test(ByRef a As Integer)
End Sub

Sub Main
    Dim v As Variant
    v = 5
    Test v                    'Visual Basic gives error here
End Sub
```

In BasicScript, the above example works as expected. BasicScript actually performs a conversion of the Variant v to a temporary Integer value and passes this temporary value by reference. Upon return from the call to Test, BasicScript converts the temporary Integer back to a Variant.

Passing Optional Variants to Forward-Declared Routines

BasicScript does not catch the following error:

```
Declare Sub Test(Optional v As Variant)          'LINE 1
Sub Main
    Test
End Sub

Sub Test(v As Variant)                           'LINE 5
End Sub
```

In the above script, the Declare statement on line 1 defines a prototype for the Test function that is incompatible with the actual declaration on line 5.

Stack Size

BasicScript uses a default stack of 2K, expandable to 8K. Visual Basic use a much larger stack size (approximately 48K).

Since the stack for BasicScript is smaller, you may have to be more attentive when using local variables, especially fixed-length strings and structures, since storage for all local variables comes from the stack.

Note: Variable-length strings only require 2 bytes of storage on the stack. Wherever possible, use variable-length strings in place of fixed-length strings.

Expression Evaluation

With Boolean expressions (i.e., expressions involving And, Or, Xor, Eqv, and Imp), if one operand is Null and the other argument is numeric, then Null is returned regardless of the value of the other operand. For example, the following expression returns Null:

```
Null And 300000
```

Despite the fact that the expression returns Null, Visual Basic evaluates the numeric operand anyway, converting it to a Long. If an overflow occurs during conversion, a trappable runtime error is generated. In BasicScript, the expression returns Null regardless of the value of the numeric operand. For example, the following expression will overflow in Visual Basic but not in BasicScript:

```
Null And 5E200
```

File Searching

The filename-matching algorithm used by BasicScript is different from that used by Visual Basic. This affects commands that perform directory searching, such as Dir, Kill, and FileList. The following differences exist:

- In Visual Basic, an asterisk within the filename matches any characters up to the end of the filename or up to the period, whichever comes first.
- In Visual Basic, the period is a separator between the filename and the extension. In BasicScript, the period is treated as a normal filename character.

The following table describes the meaning of some common file specifications.:

Specification	Meaning in Visual Basic	Meaning in BasicScript
*	All files.	All files.
.	All files.	All files that have an extension.

Specification	Meaning in Visual Basic	Meaning in BasicScript
s*e	All files that begin with "s".	All files that begin with "s" and end with "e".
s*.*	All files that begin with "s".	All files that begin with "s" and have an extension.
test.	The file "test" with no extension.	The file called "test.". BasicScript will never find this file under Windows or DOS.
test.*	All files having the root name "test" with any extension, such as "test", "test.txt", and so on.	All files having the root name "test" with an extension. The file "test" with no extension will not be found.

This filename-matching algorithm is the same across all platforms that support BasicScript.

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y