Rational Software Corporation

Extensibility Interface Reference

RATIONAL ROSE® REALTIME

VERSION: 2003.06.00

PART NUMBER: 800-026116-000

WINDOWS/UNIX

Rational®
the software development company

support@rational.com
http://www.rational.com

Legal Notices

Part Number: 800-026116-000
Version Number: 2003.06.00

from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Contents

Preface xv
Audience xv
Other Resources xv
Rational Rose RealTime Integrations With Other Rational Products xvi
Contacting Rational Customer Support xvii
1 Concepts 1
Overview 1
The RRTEI Model and Rational Rose RealTime Extensibility 2
Scripting 3
Automation 3
Type Libraries 3
About Default Properties and Property Sets (Extensibility) 4
About Collection Attributes and Operations 4
Rational Rose RealTime Menu Extensibility 6
2 How To... 7
Customizing Rational Rose RealTime Menus 7
Creating New Rational Rose RealTime Scripts 13
Getting the Rational Rose RealTime Application Object 14
Specifying a Virtual Path for Scripts 15
Working with Rational Rose RealTime Diagrams 17
Working with Model Properties 17
Working with Collections 29
Working with Classes 32
Working with Rose RealTime Automation 32
Working with the Rational Rose RealTime Script Editor 33
Opening a Model 66
Modifying a Property Value 67
Setting the Top Capsule of a Component 69

3 Rational Rose RealTime Extensibility Interface Reference 75
Logical Package Structure 80
Application Classes 81
AddIn 86
AddInManager 93
Application 93
ContextMenuItem 122
MenuState 123
PathMap 124
RsMenuState 127
Workspace 128
Extensibility Classes 130
Collection 131
RoseBase 139
RRTEIObject 140
RichTypes 141
RichType 142
RichTypeValuesCollection 144
Model Classes 145
Component View Classes 145
Component 149
ComponentPackage 170
Core Model Classes 178
ControllableElement 184
DefaultModelProperties 194
Element 204
ExternalDocument 215
Model 218
ModelElement 236
Package 239
Property 243
RsExternalDocumentType 244

StructuredProperty 244
Deployment View Classes 246
ComponentInstance 249
DeploymentPackage 252
Device 258
Processor 262
Logical View Classes 267
LogicalPackage 269
Association Classes 288
Association 290
AssociationEnd 294
AssociationEndContainment 298
AssociationEndVisibilityKind 299
Classifier Classes 299
Capsule 303
Class 304
ClassConcurrency 310
ClassKind 310
Classifier 310
ClassifierVisibilityKind 327
Parameter 328
Protocol 329
RsClassKind 332
RsConcurrency 334
RsChangeable 334
Signal 335
Feature Classes 336
Attribute 338
AttributeContainment 340
AttributeVisibilityKind 340
Operation 340
OperationConcurrency 345

OperationVisibilityKind 345
OwnerScope 346
RsOwnerScope 346
Collaboration Classes 347
AssociationEndRole 350
AssociationRole 351
CapsuleRole 352
CapsuleStructure 353
ClassifierRole 356
Collaboration 358
Connector 364
Genericity 367
Port 367
PortRole 369
PortVisibilityKind 370
RegistrationMode 370
RsGenericity 370
RsRegistrationMode 371
Common Logical View Enumerations 372
RsContainment 372
RsVisibilityKind 373
Interaction Classes 374
Environment 376
Interaction 376
InteractionInstance 382
Message 385
MessageEnd 386
RsActionKind 387
State Machine Classes 387
RsSourceRegionType 388
SourceRegionType 389
StateMachine 389

Transition 390
Action Classes 393
Action 396
ActionMode 398
CallAction 399
Coregion 399
CreateAction 401
DestroyAction 401
LocalState 402
ReplyAction 402
RequestAction 402
ResponseAction 403
ReturnAction 404
RsActionMode 404
RsSendActionPriority 405
SendAction 406
SendActionPriority 407
TerminateAction 407
UninterpretedAction 407
Event Classes 407
Event 409
EventGuard 409
PortEvent 411
ProtocolRoleEvent 415
State Classes 416
ChoicePoint 418
CompositeState 419
FinalState 424
InitialPoint 425
JunctionContinuationMode 425
JunctionPoint 425
RsJunctionContinuationMode 427

RsStateKind 427
StateKind 428
StateVertex 429
Relation Classes 431
ClassDependency 433
ClassRelation 434
ComponentDependency 435
Generalization 436
GeneralizationVisibilityKind 438
InstantiateRelation 438
LogicalPackageDependency 439
RealizeRelation 440
Relation 442
UsesRelationVisibilityKind 444
Use Case View Classes 444
UseCase 445
View Classes 450
AnchorNoteView 453
Diagram 454
NoteView 464
RsNoteViewType 466
RsStereotypeDisplay 466
StereotypeDisplay 467
ViewElement 467
Class Diagram Classes 475
CapsuleView 477
ClassDiagram 477
ClassView 490
ClassifierView 490
ProtocolView 492
Collaboration Diagram Classes 493
CapsuleRoleView 494

CollaborationDiagram 496
PortRoleView 499
PortView 500
StructurePerimeterView 500
Component Diagram Classes 501
ComponentDiagram 502
ComponentPackageView 508
ComponentView 509
Deployment Diagram Classes 509
DeploymentDiagram 510
Sequence Diagram Classes 513
ClassifierRoleView 514
CreateMessageView 514
InteractionInstanceView 515
LifeLineView 515
MessageView 516
SequenceDiagram 516
State Diagram Classes 517
BranchPointView 519
ChoicePointView 519
CompositeStateView 521
CoregionView 522
FinalStateView 522
InitialPointView 523
JunctionAdornmentView 523
JunctionPointView 524
LocalStateOrActionView 525
StateDiagram 525
StatePerimeterView 527
View Property Classes 528
LineVertex 529
View_FillColor 530

View_Font 531
View_LineColor 532
4 BasicScript Reference 535
Special Characters 536
Directives 573
Functions 578
Keywords 851
Methods 857
Operators 888
Properties 913
Statements 949
Picture Caching 1030
Optional Parameters 1061
Arrays (topic) 1198
Comments (topic) 1201
Constants (topic) 1205
Cross-Platform Scripting (topic) 1214
Dialogs (topic) 1219
Error Handling (topic) 1220
Expression Evaluation (topic) 1221
Keywords (topic) 1223
Line Numbers (topic) 1225
Literals (topic) 1225
Named Parameters (topic) 1227
Objects (topic) 1228
Operator Precedence (topic) 1231
Operator Precision (topic) 1232
User-Defined Types (topic) 1232
Index 1235

Figures

Figure 1 Rational Rose extensibility interface components 2
Figure 2 Portion of a Rational RoseRT menu file 9
Figure 3 Virtual Path Map 16
Figure 4 Specification Editor 19
Figure 5 Rose RealTime Script Editor 33
Figure 6 Goto Line dialog 35
Figure 7 Selected Scripts Text 37
Figure 8 Find Script Text dialog\ 39
Figure 9 Replace dialog 40
Figure 10 Script Calls dialog 42
Figure 11 Add Watch dialog 44
Figure 12 Modify Variable dialog 47
Figure 13 Grid Dialog 49
Figure 14 Dialog Edition with Grid Displayed 50
Figure 15 Capturing a Dialog 52
Figure 16 Sample Dialog in Basic Script 54
Figure 17 Dialog Information Dialog 61
Figure 18 Control Information dialog 62

Preface

The information in this document supersedes all other manuals and documentation included in this release.

This manual is organized as follows:

- *Concepts* on page 1
- *How To...* on page 7
- *Rational Rose RealTime Extensibility Interface Reference* on page 75
- *BasicScript Reference* on page 535

Audience

This guide is intended for all readers including managers, project leaders, analysts, developers, and testers.

This guide is specifically designed for software development professionals familiar with the target environment they intend to port to.

Other Resources

- Online Help is available for Rational Rose RealTime.

  Select an option from the **Help** menu.

  All manuals are available online, either in HTML or PDF format. To access the online manuals, click **Rational Rose RealTime Documentation** from the **Start** menu.
- To send feedback about documentation for Rational products, please send e-mail to `techpubs@rational.com`.
- For more information about Rational Software technical publications, see: `http://www.rational.com/documentation`.

- For more information on training opportunities, see the Rational University Web site: `http://www.rational.com/university`.
- For articles, discussion forums, and Web-based training courses on developing software with Rational Suite products, join the Rational Developer Network by selecting **Start > Programs > Rational Suite > Logon to the Rational Developer Network.**

Rational Rose RealTime Integrations With Other Rational Products

Integration	Description	Where it is Documented
Rose RealTime–ClearCase	You can archive Rose RT components in ClearCase.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–UCM	Rose RealTime developers can create baselines of Rose RT projects in UCM and create Rose RealTime projects from baselines.	▪ *Toolset Guide: Rational Rose RealTime* ▪ *Guide to Team Development: Rational Rose RealTime*
Rose RealTime–Purify	When linking or running a Rose RealTime model with Purify installed on the system, developers can invoke the Purify executable using the **Build > Run with Purify** command. While the model executes and when it completes, the integration displays a report in a Purify Tab in RoseRealTime.	▪ Rational Rose RealTime Help ▪ *Toolset Guide: Rational Rose RealTime* ▪ *Installation Guide: Rational Rose RealTime*
Rose RealTime–RequisitePro	You can associate RequisitePro requirements and documents with Rose RealTime elements.	▪ *Addins, Tools, and Wizards Reference: Rational Rose RealTime* ▪ *Using RequisitePro* ▪ *Installation Guide: Rational Rose RealTime*
Rose RealTime–SoDa	You can create reports that extract information from a Rose RealTime model.	▪ *Installation Guide: Rational Rose RealTime* ▪ *Rational SoDA User's Guide* ▪ SoDA Help

Contacting Rational Customer Support

If you have questions about installing, using, or maintaining this product, contact Rational Customer Support.

Your Location	Telephone	Facsimile	E-mail
North, Central, and South America	+1 (800) 433-5444 (toll free) +1 (408) 863-4000 Cupertino, CA	+1 (781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 20 4546-200 Netherlands	+31 20 4546-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Customer Support, please be prepared to supply the following information:

- Your name, company name, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your Service Request number (SR#) if you are following up on a previously reported problem

When sending email concerning a previously-reported problem, please include in the subject field: "[SR#XXXXX]", where XXXXX is the Service Request number of the issue. For example, "[SR#0176528] - New data on rational rose realtime install issue ".

Concepts 1

Contents

This chapter is organized as follows:

- *Overview* on page 1
- *The RRTEI Model and Rational Rose RealTime Extensibility* on page 2
- *Scripting* on page 3
- *Automation* on page 3
- *Type Libraries* on page 3
- *About Default Properties and Property Sets (Extensibility)* on page 4
- *About Collection Attributes and Operations* on page 4
- *Rational Rose RealTime Menu Extensibility* on page 6

Overview

Rational Rose RealTime provides several ways for you to extend and customize its capabilities to meet your specific software development needs. You can:

- Customize Rational Rose RealTime menus
- Automate manual Rational Rose RealTime functions with Rational Rose RealTime Scripts (for example, diagram and class creation, model updates, document generation, etc.)
- Execute Rational Rose RealTime functions from within another application by using the Rational Rose RealTime Automation object.
- Access Rational Rose RealTime classes, properties and methods right within your software development environment by including the Rational Rose RealTime Extensibility Type Library in your environment.
- Use the Add-In Manager

The RRTEI Model and Rational Rose RealTime Extensibility

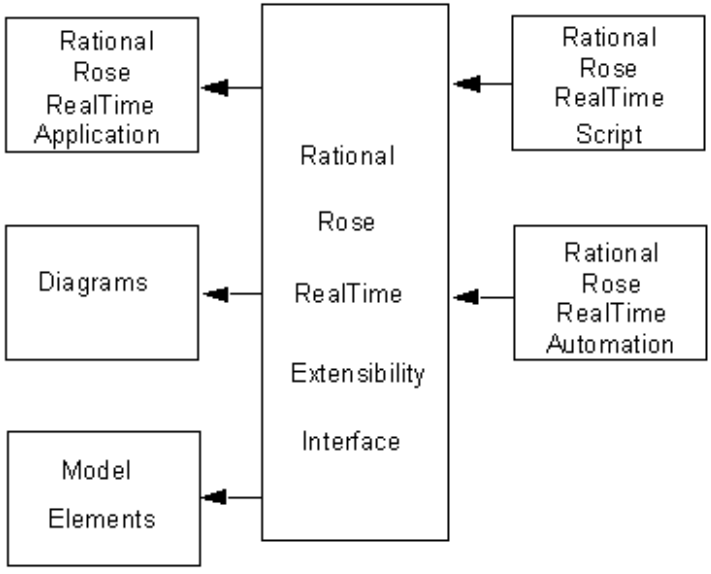
The purpose of Rational Rose RealTime is to enable component-based software development. As you would expect, the Rational Rose RealTime application is itself component-based, and is defined in the Rational Rose RealTime Extensibility Interface (RRTEI) Model.

The RRTEI Model is essentially a metamodel of a Rational Rose RealTime model, exposing the packages, classes, properties and methods that define and control the Rational Rose RealTime application and all of its functions.

You communicate with the Rational Rose RealTime Extensibility Interface through Rational Rose RealTime Scripts or through Rational Rose RealTime Automation. In either case, you will use the RRTEI calls defined in the Rational Rose RealTime Extensibility Interface Reference. This reference is available in printed form, and is also part of this online help.

Figure 1 shows the core Rational Rose RealTime components, the Rational Rose extensibility interface components, and the relationships between them.

Figure 1 Rational Rose extensibility interface components



Scripting

The Rational Rose RealTime Scripting language is an extended version of the Summit Basic Scriptlanguage. The Rational Rose RealTime extensions to basic scripting allow you to automate Rational Rose RealTime-specific functions, and in some cases perform functions that are not available through the Rational Rose RealTime user interface.

The Rational Rose RealTime script editor runs in the Rational Rose RealTime environment and provides your access to the scripting environment. Start the script editor by selecting either New Script or Open Script from the Tools menu.

Automation

Rational Rose RealTime automation allows you to integrate other applications with Rational Rose RealTime in two ways:

- Using Rational Rose RealTime as an automation controller, you can call an OLE automation object from within a Rational Rose RealTime script. For example, a Rational Rose RealTime script can use OLE automation to execute functions in applications such as Word and Excel.
- Using Rational Rose RealTime as an automation server, you can call its OLE automation object from within other OLE-compliant applications.

Rational Rose RealTime Automation is accessible to automation controller environments such as Visual Basic, Summit BasicScript, Softbridge Basic Language, Visual C++, and others.

Use the online BasicScript and Rational Rose RealTime Script Language References for complete script language information.

Type Libraries

Loading a type library for Rational Rose RealTime automation allows you to use Rational Rose RealTime class names to access the Rational Rose RealTime Extensibility Interface from your programming environment.

For example, if you are working in Visual Basic, instead of using the Basic object type **Object**, you can use the name of the actual Rational Rose RealTime class. You can also check the syntax of the properties and methods at compile time (early binding) instead of when the code is executed (late binding).

If you are working in Visual C++, you can import Rose RealTime's type library, which is embedded in RrtRes.dll, into an MFC project. This generates **COleDispatchDriver** subclasses for each RRTEI class, and methods allowing access to RRTEI properties and methods.

About Default Properties and Property Sets (Extensibility)

Each Rational Rose RealTime model has its own default properties. These default properties are defined in a property file and are grouped into sets based on:

- Type of model element

  Class, component, relation, attributes, operations; and so on; the objects that make up the model

- Tool

  Corresponds to a tab in the property specification. A tool can be a programming language tool (such as C++), a user-defined add-in to Rational Rose RealTime, or some other tool.

- Properties

  The actual properties and property values defined in the set; these must be appropriate to the model element and tool for which they are being defined.

Note: You can define multiple sets of default properties for the same tool and model element. For example, you might want one set of properties for a class with a stereotype of Actor and a different set of properties for a class with a stereotype of Interface. Both of these sets are considered default properties in that they are predefined for the model. Defining multiple sets saves you work by minimizing the need to override properties as you go.

About Collection Attributes and Operations

For most elements of a Rose RealTime model there is a corresponding collection. So, for example, for every class there is a class collection; for every logical package there is a logical package collection; for every property, there is a property collection, and so on.

Rational Rose RealTime extensibility provides a set of properties and methods that allow you to access a particular element in any given collection.

Collection Property

Count is the only property that applies to collections.

Count - Number of objects within a collection

Methods for All Collections

The following table describes the collection methods that allow you to locate and retrieve the elements in any collection. While all of these properties and methods are the same, they act upon different types of objects. For example, the **ClassCollection.GetAt** method retrieves a class object, the **LogicalPackageCollection.GetAt** method retrieves a logical package, and so on.

Method	Description
Exists	Indicates whether an object exists in a given collection
FindFirst	Retrieves the index (position) of the first instance of an object in a given collection
FindNext	Retrieves the index (position) of the next instance of an object in a given collection
GetWithUniqueID	Retrieves the instance of an object in a given collection, given the object's unique ID ***Note:*** *Objects that do not have a uniqueID (for example, ExternalDocument and Property objects) cannot be retrieved using this method.*
GetAt	Retrieves a specified instance of an object in a given collection
GetFirst	Retrieves the first instance of an object from a given collection
GetObject	Returns the OLE interface object associated with the given collection
IndexOf	Finds the index (position) of an object in a given collection

Methods for User-defined Collections

The following table describes the four additional collection methods, which allow you to add and remove objects from a collection. However, these methods are only valid for user-defined collections and cannot be used with Rose RealTime Model collections:

Method	Description
Add	Adds an object to the object collection
AddCollection	Adds a collection to an object collection
Remove	Removes a collection from an object collection
RemoveAll	Removes the entire contents of a collection

User-defined collections are created by the CreateCollection function of the Rational Rose RealTime Application object.

Rational Rose RealTime Menu Extensibility

You extend, or customize, Rational Rose RealTime menus by updating the Rational Rose RealTime menu file, **rosert.mnu**, which Rational Rose RealTime reads during startup.

You can extend Rational Rose RealTime menus by adding:

- Submenus
- Menu options that execute any of the following:
    - Rational Rose RealTime primitives
    - Rational Rose RealTime scripts
    - System commands
    - External programs
- Menu separators (lines between menu options, used to group similar menu items)

Note: You can add information to existing menus (for example, File, Edit, etc.); however, you cannot add new menus to the Rational Rose RealTime menu bar.

How To... 2

Contents

This chapter is organized as follows:

- *Customizing Rational Rose RealTime Menus* on page 7
- *Creating New Rational Rose RealTime Scripts* on page 13
- *Getting the Rational Rose RealTime Application Object* on page 14
- *Specifying a Virtual Path for Scripts* on page 15
- *Working with Rational Rose RealTime Diagrams* on page 17
- *Working with Model Properties* on page 17
- *Working with Collections* on page 29
- *Working with Classes* on page 32
- *Working with Rose RealTime Automation* on page 32
- *Working with the Rational Rose RealTime Script Editor* on page 33
- *Opening a Model* on page 66
- *Modifying a Property Value* on page 67
- *Setting the Top Capsule of a Component* on page 69

Customizing Rational Rose RealTime Menus

The content of Rational Rose RealTime menus is defined in the **rosert.mnu** file. If you want to customize Rational Rose RealTime menus, you must edit this file.

While you cannot add new menus to the Rational Rose RealTime menu bar, you can add commands to the existing Rational Rose RealTime menus. The menu actions defined for the Rational Rose RealTime menu file allow you to add commands that:

- Execute a program or shell script
- Execute a Rational Rose RealTime script
- Display a dialog for user input

To customize Rational Rose RealTime menus:

1. Using any text editor, open the **rosert.mnu** file.
2. Add entries to **rosert.mnu** for any or all of the following:
   - Submenus
   - Menu options
   - Menu separators

   Ensure that you follow the appropriate syntax rules as you add the entries in the file.
3. If your menu item executes a script, add or edit Rational Rose RealTime's virtual path for scripts (if one is not already defined).
4. Save the file:
   - To create another menu file while leaving **rosert.mnu** intact, save the file under a different name. (Recommended)
   - To overwrite the file, save it as **rosert.mnu.**

Adding Entries to a Rational Rose RealTime Menu File

Using any text editor and the following information, you can add menu entries to the Rational Rose RealTime menu file. The entries appear on the Rational Rose RealTime menu in the order you specify.

As you add menu entries, you specify:

- Keywords that determine what to add to the menu (a submenu, a menu option, a separator)
- Arguments that further define a menu action, or that determine the conditions under which a menu action command is enabled or disabled in Rational Rose RealTime.
- Menu actions that specify what action occurs when the menu item is selected.

Pay close attention to the syntax rules that apply to your entries to the Rational Rose RealTime menu file. For example, the syntax of the menu specifications includes opening and closing braces. You must include these braces in your specifications for them to work properly. Remember that each opening brace ({) requires a corresponding closing brace (}).

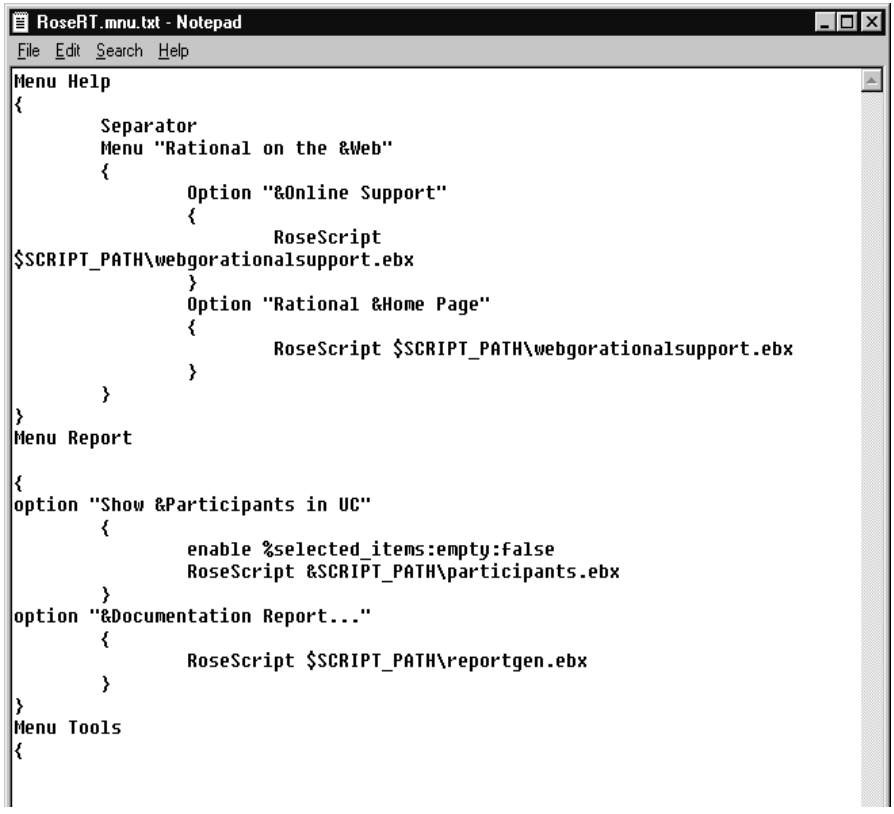
Creating a New Rational Rose RealTime Menu File

The best way to create a new Rational Rose RealTime menu file is to save an existing menu file using a new name. This keeps the existing file intact, while providing a complete menu file to make changes.

Sample Rational RoseRT Menu File

The following example shows a portion of a Rational RoseRT menu file.

Figure 2 Portion of a Rational RoseRT menu file



Syntax Rules for Rational Rose RealTime Menu File Entries

Follow these rules when specifying menu text:

- When a text string contains embedded spaces, enclose the string in double quotation marks.

Example: “Run Script”

- When a text string has no embedded spaces (a single word, for example), enter the string without any quotation marks.

Example: Validate

- When a text string that is not enclosed in quotes includes a special character, the special character could be misinterpreted as a variable. For this reason, you must precede any special characters (such as ^, ì, or %) with an escape character. The escape character for all special characters is ^.

Examples:

Option Calculate^% creates a menu option whose text reads Calculate %

exec Notepad ^””c:\my files\file.txt”^” creates a menu action that executes the following command line: notepad “c:\my files\file.txt” Note the escape character followed by an additional set of quotation marks. One set of quotation marks is necessary because there is a space in my files. The second set, each of which is preceded by the ^ escape character, causes the actual command line to include the quotation marks as part of the command.

- To create a mnemonic for the menu, add an & before the menu text.

Example: “&Run Script”

Allows users to execute the menu item by entering CTL+R

- Menu text can include *Variables* on page 12 and *Modifiers* on page 11

Example: Option “Validate “%model

Creates a menu option with the text Validate MyModel if the currently loaded model is MyModel.mdl.

Menu File Keywords

Valid keywords for your entries to the Rational Rose RealTime menu file are described below:

- **Menu RoseRTMenu** - Enter the Menu keyword, followed by the Rational Rose RealTime menu name to indicate the name of the menu being extended. Example: Enter Menu Tools as the first line of an entry that extends the Tools menu.
- **Menu "Menu Text"** - Enter the Menu keyword, followed by a text string to indicate the name of a submenu being added to the menu. Note that quotation marks are required if the text string contains spaces. Example: Enter Menu “RoseRT Scripts” to add a submenu called RoseRT Scripts.
- **Separator -** Enter the Separator keyword to add a separator to a list of menu options. Remember the placement of the Separator keyword controls the placement of the separator line on the menu.
- **Option "Command text" -** Enter the Option keyword, followed by a text string to indicate the name of the menu command being added to the menu. Note that quotation marks are required if the text string contains spaces. Example: Enter Option “Run My Script” to add a menu command called Run My Script.

Modifiers

Rational Rose RealTime provides a set of *Variables* on page 12 that correspond to various Rational Rose model items. You can use these variables in conjunction with a set of *Modifiers* on page 11 to determine the conditions under which menu items are enabled or disabled, as well as to specify specific menu actions.

The format for specifying variables with modifiers is:

```
variable[:mod1[:mod2[...[:mod10]]]
```

Modifiers

The modifiers [cmumod.cpp] are:

- :not
- :writeable
- :home_unit
- :empty
- :unary
- :first
- :file
- :basename
- :directory

- :elide
- :codefile
- :headerfile
- :sourcefile
- :allfiles
- :multiple

Variables

The variables [cmuvar.cpp] are:

- %current_diagram
- %selected_items
- %model
- %selected_units
- %all_units
- %false
- %true

Menu Actions

An action defines the result of activating a menu entry. The required arguments can be supplied as constants, variables, or *Variables* on page 12 with *Modifiers* on page 11.

- **Block** - Displays a modal dialog with ‘arg’ as its prompt. Used following ‘exec’ and an action to suspend the following action until the user chooses to continue
- **Rosescript** - Executes a source or compiled image of a script. You can specify the script name without its extension. The Rosescript command will search for the source script first and execute it if found. If not found, it will search for and execute the compiled script.
- **Exec pathname [arg2 [arg3 ...[arg10]]]** - Executes the program or shell script contained in the file designated by program-name. (If the program is not located in the current directory, it must be in a directory in the execute path.) If the final argument is of the form 'F<filename>' then a file named <filename> is created (if it does not already exist). All arguments, except the last one are written to the file, and <filename> is passed as the sole argument to the program.

  Notes:

  - F must be uppercase.
  - It is up to ‘program’ to delete the file
  - To pass a string beginning with ‘-F’ as the final parameter of an exec action, use ‘--F’. (The character ‘^’ does NOT work in this case.)

Adding Scripts to a Rational Rose RealTime Menu

To add a script to a Rational RoseRealTime menu:

1. Open the Rational Rose RealTime Menu file, or create a new one to use in its place.
2. Edit the Path Map so that it includes a virtual script path.
3. Modify the Rational Rose RealTime menu file to add the script under the appropriate menu, being careful to follow all of the menufile syntax rules. To do this:

- In the menu file, locate the menu specification that corresponds to the Rational Rose RealTime menu to which you want to add the script. Each menu specification is comprised of the Menu keyword followed by the name of a Rational Rose RealTime menu. For example, the Tools menu specification begins with **Menu Tools**.
- Within the appropriate menu specification, add a menu option that specifies the text of the menu command that will run the script (for example, "Run Conversion Wizard")
- Enter a Rational RoseScript menu action to cause the script to execute when a user selects the menu command.

4. Save the updated menu file.

Creating New Rational Rose RealTime Scripts

To create a new Rational Rose RealTime script:

1. Select **Tools/New Script** from the Rational Rose RealTime menu bar.
2. Enter your script text.
3. Select **File/Save As** from the Rational Rose RealTime menu bar and save the new script.

Creating a New Script from an Existing Script

To modify an existing script:

1. Select **Tools/Open Script** from the Rational Rose RealTime menu bar.
2. Select a file from the list of available scripts

3 Click **OK** to enter the script editor and display the script.

4 Select **File/Save As** from the Rational Rose RealTime menu bar and save the new script.

Getting the Rational Rose RealTime Application Object

Whether you are using Rational Rose RealTime Script or Rational Rose RealTime Automation, you must get the Rational Rose RealTime Application object in order to control the Rational Rose application.

Using Rational Rose RealTime Script

All Rational Rose RealTime Script programs have a global object called RoseRTApp, which represents the Rose RealTime Application object.

Using Rational Rose RealTime Automation

To use Rational Rose RealTime as an automation server, you must initialize an instance of a Rational Rose RealTime application object. You do this by calling either CreateObject or GetObject (or their equivalents) from within the application you are using as the OLE controller.

These calls return the OLE Object which implements Rational Rose RealTime API's application object.

Refer to the documentation for the application you are using as OLE controller for details on calling OLE automation objects.

RoseRTApp.CurrentModel Example (Scripting)

The following sample code shows how to get the Rational Rose RealTime application object in a Rational Rose RealTime Scripting context:

Sub GenerateCode (theModel As RoseRTModel)

```
'This generates code
End Sub
Sub Main
GenerateCode RoseRTApp.CurrentModel
End Sub
```

RoseRTApp.CurrentModel Example (Automation)

The following sample code shows how to get the Rational Rose RealTime application object in a Rational Rose RealTime Automation context:

Sub GenerateCode (theModel As Object)

```
'This generates code
End Sub
Sub Main
Dim RoseRTApp As Object
Set RoseRTApp = CreateObject
(“RoseRT.Application”)
GenerateCode RoseRTApp.CurrentModel
End Sub
```

A Polling Add-In (automation)

A polling add-in can make calls to sleep and do events, thus interacting with the toolset at the same time that this script is running. Note, however, that the toolset has a visible state accessible from the Application object. If you exit the toolset and at least one Add-In is still running, the toolset becomes invisible and runs in the background until the Add-In releases its application pointer.

Specifying a Virtual Path for Scripts

Adding or Editing the Virtual Path for Scripts

When you edit the Rational Rose RealTime menu file to include script commands, you must include one of the following:

- The fully qualified name of the script file to execute
- The virtual path that maps to the actual path

Defining a virtual path for scripts simplifies the process of editing the menu file by allowing you to specify the symbolic virtual path name instead of the complete file path.

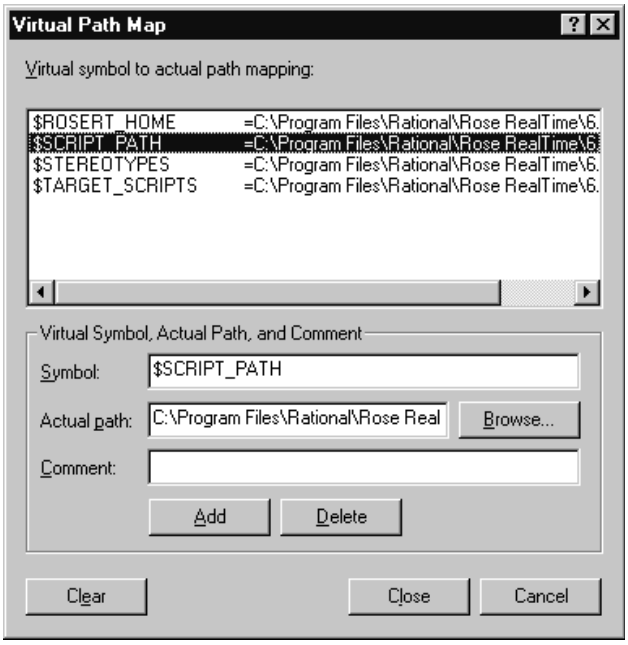
To add or edit a virtual path for scripts:

1. Start Rational Rose RealTime.
2. Select Edit Path Map from the **File** menu to display the Virtual Path Map dialog.

3 Check for the **$SCRIPT_PATH** virtual symbol and do one of the following:

- If the symbol exists, select it in the dialog to display its current mapping information in the lower portion of the dialog.
- If the symbol does not exist, enter it in the Symbol field in the lower portion of the dialog.

Figure 3 Virtual Path Map



4 Enter the actual path to your Rational Rose RealTime scripts, or use the **Browse** button to locate and select the path. (Normally these scripts reside in a Scripts subdirectory of the Rational Rose RealTime installation directory.) Press **Add**.

5 When you make changes in the dialog, the **Close** button becomes an **OK** button. Select **OK** to save your changes and exit the Virtual Path Map.

Working with Rational Rose RealTime Diagrams

Each kind of Rational Rose RealTime diagram (class, component, scenario, etc.) inherits from the Diagram class.

A diagram is made up of ModelElements and ViewElements. A ViewElement is the physical representation of the actual Rose RealTime Model Element. As such, it is an object with properties and methods that define its appearance in the diagram window (position, color, size, etc). You can define multiple ViewElements for any given ModelElement.

- Use Diagram.ViewElements to iterate through the collection of ViewElements belonging to a diagram.
- Use Diagram.ModelElements to iterate through the ModelElements that exist in the diagram.
- Use Diagram.GetViewFrom to find the first ViewElement of a given ModelElement.

Note: You can only use GetViewFrom to retrieve the first ViewElement defined for the ModelElement. Even if you have more than one view, you'll always only get the first.

- To find out which ViewElements are currently selected in a diagram, iterate through the diagram's ViewElements. As you retrieve each ViewElement, use the ViewElement.IsSelected method to find out whether it is currently selected in the diagram. You can then retrieve the selected ModelElement, or do any other processing you want to do based on whether ViewElement is selected.
- A short way to retrieve all selected ModelElements from a diagram is to use the Diagram.GetSelectedModelElements method. Instead of iterating through the diagram and checking each ViewElement, this method simply returns everything that is selected.

Working with Model Properties

Working with model properties includes

- *Managing Default Properties (Extensibility)* on page 18
- *Creating a New Property* on page 20
- *Deleting Model Properties* on page 20
- *Creating a New Property Set* on page 20
- *Getting and Setting the Current Property Set* on page 21
- *Getting Model Properties* on page 22
- *Deleting a Model Property* on page 22

- *Adding a Property to a Set* on page 23
- *Creating a User-Defined Property Type* on page 24
- *Cloning a Property Set* on page 25
- *Setting Model Properties* on page 27
- *Setting Model Properties Using InheritProperty* on page 27
- *Setting Model Properties Using OverrideProperty* on page 28
- *Creating a New Tool* on page 29

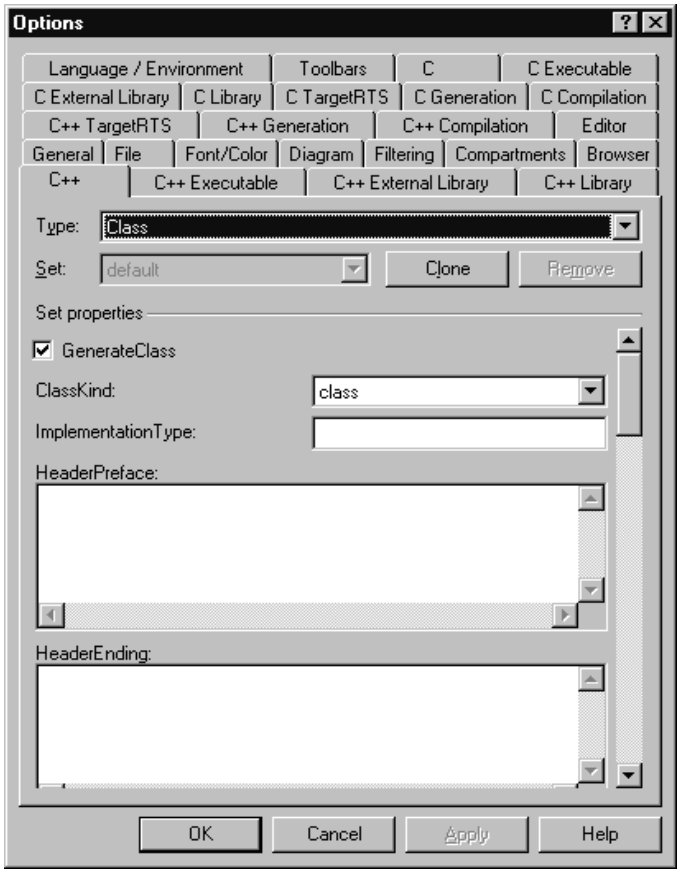
Managing Default Properties (Extensibility)

In the Rational Rose RealTime user interface environment, you manage a model's properties by using the specification editor.

To access the specification editor, you point to **Model Properties** on the **Tools** menu and select **Edit**.

You then select the appropriate tool tab, element type, and property set to edit. For example, in the following figure, the tool is C++, the model element type is **Class**, and the property set is **default**.

Figure 4 Specification Editor



From this point on, you can use the specification editor to edit individual properties, as well as clone (copy) and edit property sets. However, you cannot create new tools (tabs), new default property sets, or property types. For these capabilities, you must use the Rational Rose RealTime Extensibility Interface.

For more information on editing default properties and sets in the Rational Rose RealTime user interface, check the online help for information on Specifications.

In the Extensibility Interface, the DefaultModelProperties object manages the default model properties for the current model, and is itself a property of the model (RoseRTApp.CurrentModel.DefaultProperties). For this reason, default properties are applied to the current model only. When you create default properties they are applied and saved for the current model, but are not available to any new models you create.

To apply new properties to another model, re-run the script that creates the properties, specifying the new model as the current model.

Creating a New Property

How To

To create a new property that is not based on an existing property, use the CreateProperty method. However, if you simply want to set an existing property to a different current value, you should use InheritProperty or OverrideProperty instead.

Example

```
' Property creation:


b = theModel.RootLogicalPackage.CreateProperty (myTool, "Saved",
"True", "Boolean")


' Property destruction:
b = theModel.RootLogicalPackage.InheritProperty (myTool, "Saved")
```

Notes on the Example

1. The CreateProperty call in the example creates a new property called **Saved**. It applies to the tool **MyTool**, its value is **True** and its type is **Boolean**.
2. The InheritProperty call in the example deletes the property just created.

Deleting Model Properties

If you are deleting a property that belongs to a property set, you can use the DeleteDefaultProperty method to delete the property from a model.

However, if you created a property using the CreateProperty method, that property is not part of a property set. To delete such a property, use the InheritPropertymethod.

Creating a New Property Set

To create a new property set from scratch, use the CreateDefaultPropertySet method.

Getting and Setting the Current Property Set

How To

To find out the which property set is the current set for a tool, use the GetCurrentPropertySetName method.

To set the current property set to a particular set name, use the SetCurrentPropertySetName to the set of your choice.

Note: When setting the current property set, you must supply a set name that is valid for the specified tool. To retrieve a list of valid set names for a tool, use the GetDefaultSetNames Method (Element).

Example

```
Sub RetrieveElementProperties (theElement As RoseRT.Element)
   Dim AllTools As RoseRT.StringCollection
   Dim theProperties As RoseRT.PropertyCollection
   Dim theProperty As RoseRT.Property
   Set AllTools = theElement.GetToolNames ()
   For ToolID = 1 To AllTools.Count
      ThisTool$ = AllTools.GetAt (ToolID)
      theSet$ = theElement.GetCurrentPropertySetName (ThisTool$)
      Set theProperties = theElement.GetToolProperties (ThisTool$)
      For PropID = 1 To theProperties.Count
         Set theProperty = theProperties.GetAt (PropID)
      Next PropID
   Next ToolID
End Sub
```

Notes on the Example

1. GetToolNames retrieves the tool names that apply to the model element type called **Element** and returns them as a string collection called **AllTools**.
2. The current property set is retrieved for each tool name.
3. GetToolProperties retrieves the property collection that belongs to the current tool.
4. Each property that belongs to the tool's property collection is retrieved.

Getting Model Properties

The Element class provides two methods for retrieving information about model properties:

- To get the current value for a model property, whether inherited or overridden, use the GetPropertyValue method. This method returns the value as a string
- To retrieve the property object itself, use the FindProperty.

Deleting a Model Property

How To

To delete an entire property set from a model, use the DeleteDefaultPropertySet method.

Example

```
Sub DeleteDefaultProperties (theModel As RoseRT.Model)
   Dim DefaultProps As RoseRT.DefaultModelProperties

   Set DefaultProps = theModel.DefaultProperties
myClass$ = theModel.RootLogicalPackage.GetPropertyClassName ()

   b = DefaultProps.DeleteDefaultPropertySet (myClass$, myTool$,
"SecondSet")

   b = DefaultProps.DeleteDefaultPropertySet (myClass$, myTool$,
"ThirdSet")

   b = theModel.RootLogicalPackage.SetCurrentPropertySetName
(myTool$, "default")

End Sub
```

Notes on the Example

1. The GetPropertyClassName retrieves the valid internal class name to pass as a parameter on the delete calls.
2. Each DeleteDefaultPropertySet call deletes a property set from the model.
3. The SetCurrentPropertySetName call sets the tool's current property set its original set, which happens to be called default.

Adding a Property to a Set

To add a property to a property set, define a subroutine that uses the AddDefaultProperty method. Notice that this method requires you to pass six parameters:

- Class Name
- Tool Name
- Set Name
- Name of the New Property
- Property Type
- Value of the New Property

Example

```
Sub AddDefaultProperties (theModel As RoseRT.Model)

   Dim DefaultProps As RoseRT.DefaultModelProperties

   Set DefaultProps = theModel.DefaultProperties myClass$ =
theModel.RootLogicalPackage.GetPropertyClassName ()


   b = DefaultProps.AddDefaultProperty (myClass$,myTool$, "Set1",
"StringProperty", "String", "")

      b = DefaultProps.AddDefaultProperty (myClass$,myTool$, "Set1",
"IntegerProperty", "Integer", "0")

      b = DefaultProps.AddDefaultProperty (myClass$,
myTool$,*Set1","FloatProperty", "Float", "0")

      b = DefaultProps.AddDefaultProperty (myClass$,myTool$, "Set1",
"CharProperty", "Char", " ")

   b = DefaultProps.AddDefaultProperty (myClass$,myTool$, "Set1",
"BooleanProperty", "Boolean","True")

End Sub
```

Notes on the Example

1. When you specify the Class Name parameter, you must specify the internal name of the model element. There are two ways to obtain this information:
   - If properties are already defined for this element, it will appear in the specification dialog in the Rational Rose RealTime user interface. Simply check the specification editor and use the Type drop-down list to find the appropriate class name.
   - Use the GetPropertyClassName method. This is the method used in the sample script. This example retrieves the internal name and returns it in myClass$, which is then passed as the class name parameter.
2. If the tool you specify does not exist, a new tool will be created. This is actually the only way to add a new tool to a model.
3. This example adds a property of each of the predefined property types, except the enumeration type. You use the enumerated type to create your own property types and add enumerated properties to a set. See Creating a User-Defined Property Type for instructions and an example.

Creating a User-Defined Property Type

Rational Rose RealTime Extensibility defines a set of predefined property types. When you add properties to a set, you specify one of these types.

In addition, you can define your own property types and add properties of that type to a property set.

To create a user-defined property type, add a property whose type is enumeration and whose value is a string that defines the possible values for the enumeration.

Once you have defined the new type, adding a property of this new type is like adding any other type of property.

Example

```
Sub AddDefaultProperties (theModel As RoseRT.Model)
Dim DefaultProps As
RoseRT.DefaultModelProperties

Set DefaultProps = theModel.DefaultProperties
myClass$ =
theModel.RootLogicalPackage.GetPropertyClassName
```

```
()

b = DefaultProps.AddDefaultProperty (myClass$,
myTool$, "Set1", "MyNewEnumeration",
"Enumeration", "Value1,Value2,Value3")

b = DefaultProps.AddDefaultProperty (myClass$,
myTool$, "Set1", "MyEnumeratedProperty",
"MyNewEnumeration", "Value1")

End Sub
```

Notes on the Example

1. This example uses the GetPropertyClassName to retrieve the internal name of the class to which the property type will apply.
2. The first AddDefaultProperty call adds the enumeration and defines its possible values in the string ìValue1,Value2,Value3î.
3. The second AddDefaultProperty call adds a new property of the new enumerated type; the property value is set to ìValue1î.
4. If you want a new type to appear in the specification dialog in the Rational Rose Realtime user interface, you must actually add a property of that type to the set. Using the above example, if you simply created the type **MyNewEnumeration**, but did not add the property **MyEnumeratedProperty**, **MyNewEnumeration** would not appear in Type drop-down. Once you add the actual property, **MyNewEnumeration** would appear in the list of types.

Cloning a Property Set

How To

Cloning allows you to create a copy of an existing property set for the purpose of creating another property set. This is the easiest way to create a new property set, and is particularly useful for creating multiple sets of the same properties, but with different values specified for some or all of the properties.

To clone a property set in a model, use the CloneDefaultPropertySet method.

Example

```
Sub CloneDefaultProperties (theModel As RoseRT.Model)
   Dim DefaultProps As
   RoseRT.DefaultModelProperties

   Set DefaultProps = theModel.DefaultProperties

   AddDefaultProperties theModel

   myClass$ = theModel.RootLogicalPackage.GetPropertyClassName
()

   b = DefaultProps.CloneDefaultPropertySet (myClass$, myTool$,
"default", "SecondSet")

   b = DefaultProps.CloneDefaultPropertySet (myClass$, myTool$,
"default", "ThirdSet")

   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"SecondSet", "StringProperty", "String", "Unique to SecondSet")

   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"SecondSet", "IntegerProperty", "Integer", "11")

   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"SecondSet", "FloatProperty", "Float", "89.9000")

   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"SecondSet", "EnumeratedProperty", "EnumerationDefinition",
"Value2")

   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"ThirdSet", "StringProperty", "String", "Unique to ThirdSet")
```

```
   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"ThirdSet", "IntegerProperty", "Integer", "20")


   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"ThirdSet", "FloatProperty", "Float", "90.9000")


   b = DefaultProps.AddDefaultProperty (myClass$, myTool$,
"ThirdSet", "EnumeratedProperty", "EnumerationDefinition",
"Value3")


End Sub
```

Notes on the Example

1. This example clones an existing property set twice in order to define a total of three sets for the class and tool to which the sets apply.
2. All three sets have the same properties as those defined in the original set. In addition, several new properties are added to the second set and several other new properties are added to the third set.

Setting Model Properties

There are several ways to set model properties using the Extensibility Interface:

- Use the OverrideProperty method to change only the value of a property, and keep all other aspects of the property definition intact
- Use the InheritProperty method to return a previously overridden property to its original value
- Use the CreateProperty or the AddDefaultProperty method to define a new property from scratch

Setting Model Properties Using InheritProperty

How To

Use the InheritProperty method to reset an overridden property to its original value.

You can also use this method to delete a property that you created using the CreateProperty method. Because there is no default value to which such a property can return, InheritProperty effectively deletes it from the model.

Example

```
Sub InheritRadioProps (theLogicalPackage As RoseRT.LogicalPackage)
   b = theLogicalPackage.InheritProperty (myTool$, "StringProperty")

   b = theLogicalPackage.InheritProperty (myTool$,
"IntegerProperty")

   b = theLogicalPackage.InheritProperty (myTool$, "FloatProperty")

   b = theLogicalPackage.InheritProperty (myTool$,
"EnumeratedProperty")

End Sub
```

Notes on the Example

Each of the four lines of the sample subroutine returns the current value of the specified property to its original value.

Setting Model Properties Using OverrideProperty

How To

The OverrideProperty method allows you to use the default property definition and simply change its current value. Alternately, you could create a brand new property by calling the CreateProperty method, but that would require you to specify the complete property definition, not just the new value.

If the property you specify does not exist in the model's default set, a new property is created for the specified object only. This new property is created as a string property.

Example

```
Sub OverrideRadioProps (theLogicalPackage As RoseRT.LogicalPackage)
   b = theLogicalPackage.OverrideProperty (myTool$,
"StringProperty", "This string is overridden")

   b = theLogicalPackage.OverrideProperty (myTool$,
"IntegerProperty", "1")
```

```
   b = theLogicalPackage.OverrideProperty (myTool$, "FloatProperty",
"111.1")


   b = theLogicalPackage.OverrideProperty
(myTool$,"EnumeratedProperty", "Value2")


End Sub
```

Notes on the Example

1. Each of the four lines of the sample subroutine changes the current value of a specific property as follows:
   - The property called StringProperty now has a value of This string is overridden.
   - The property called IntegerProperty now has value of 1.
   - The property called FloatProperty now has a value of 111.1
   - The property called EnumeratedProperty now has a value of Value2.
2. Everything except for current value (tool name, class name, set, property name and property type) remains the same for the properties.

Creating a New Tool

There is no explicit way to add a new tool (tab) to a model. However, when you create a new property set or add a new property to a model, you must specify the tool to which the property or set applies. If the tool you specify does not already exist, it will be added during the create or add process.

Working with Collections

Working with collections includes

- *Getting an Element from a Collection (Overview)* on page 30
- *Accessing Collection Elements By Count* on page 30
- *Accessing Collection Elements By Name* on page 30
- *Accessing Collection Elements By Unique ID* on page 31

Getting an Element from a Collection (Overview)

There are three ways to get an individual model element from a collection:

- Use the GetwithUniqueID method to directly access the element.
- Iterate through the collection using the element's name using FindFirst, FindNext, and GetAt.
- Iterate through the collection using Count followed by GetAt.

Accessing Collection Elements By Count

How To

Follow these steps to access collection elements by count:

1. Iterate through the collection using the Count property.
2. Retrieve the specific element using the GetAt method when the specific element is found.

Example

```
Dim AllClasses As RoseRT.ClassCollection
Dim theClass As RoseRT.Class
For ClsID = 1 To AllClasses.Count
   Set theClass = AllClasses.GetAt (ClsID)
   ' ToDo: Add your code here...
Next ClsID
```

Accessing Collection Elements By Name

How To

Follow these steps to access an operation belonging to a class:

1. Use FindFirst to find the first occurrence of the specified operation in the collection.
2. Use FindNext to iterate through subsequent occurrences of the operation.
3. Retrieve the specific operation using the GetAt method when the specific operation is found.

Example

```
    Sub PrintOperations (theClass As RoseRT.Class, OperationName As
  String)

    Dim theOperation As RoseRT.Operation

    OperID = theClass.Operations.FindFirst (OperationName$)
    Do Until OperID = 0

    Set theOperation = theClass.Operations.GetAt (OperID)

    ' ToDo: Add your code here...

    OperID = theClass.Operations.FindNext (OperID, OperationName$)

    Loop

  End Sub
```

Accessing Collection Elements By Unique ID

How To

The most direct and easiest way to get an element from within a collection is by unique id. Follow these steps to access collection elements by unique ID:

1. Use the GetUniqueID method to obtain the element's unique id.
2. Use the GetwithUniqueID method, specifying the id you obtained in step 1.

Example

```
Dim theClasses As RoseRT.ClassCollection
Dim theClass As RoseRT.Class
theID=theClasses.theClass.GetUniqueID ()
theClass = theClass.GetwithUniqueID (theID)
```

Working with Classes

Placing Classes in LogicalPackages

- To create a new class and place it in a LogicalPackage, you use the AddClass method.
- To relocate an existing class from one LogicalPackage to another, use the RelocateClass method.

Working with Rose RealTime Automation

Whether you are using Rational Rose RealTime Script or Rational Rose RealTime Automation, you must get the Rational Rose RealTime Application object in order to control the Rational Rose application.

Using Rational Rose RealTime Script

All Rational Rose RealTime Script programs have a global object called RoseRTApp, which represents the Rose RealTime Application object.

Using Rational Rose RealTime Automation

To use Rational Rose RealTime as an automation server, you must initialize an instance of a Rational Rose RealTime application object. You do this by calling either CreateObject or GetObject (or their equivalents) from within the application you are using as the OLE controller.

These calls return the OLE Object which implements Rational Rose RealTime API's application object.

Refer to the documentation for the application you are using as OLE controller for details on calling OLE automation objects.

Working with the Rational Rose RealTime Script Editor

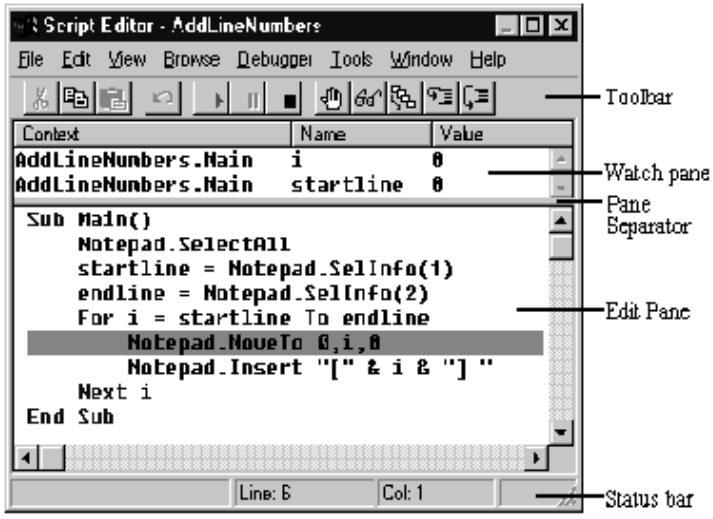
The Rose RealTime Script Editor provides your environment for creating, debugging, and compiling scripts that work with the Rose RealTime Extensibility Interface.

The Script Editor Window

As shown in Figure 5, the Script Editor's application window contains the following elements:

- Toolbar: a collection of tools that you can use to provide instructions to the Script Editor
- Edit pane: a window containing the source code for the script you are currently editing
- Watch pane: a window that opens to display the watch variable list after you have added one or more variables to that list
- Pane separator: a divider that appears between the edit pane and the watch pane when the watch pane is open
- Status bar: displays the current location of the insertion point within your script

Figure 5 Rose RealTime Script Editor



Opening a Script

To open a script in the Script Editor.

1. Click **Open Script** from the **Tools** menu.
2. Select the script to open and select **OK**.

   The script is displayed in a new Script Editor window.

Creating New Rational Rose RealTime Scripts

Creating a New Script from Scratch

To create a new script in the Script Editor.

1. Click **New Script** from the **Tools** menu.
2. Enter your script in the new Script Editor window.
3. Enter your script text.
4. Click **Save Script** from the **File** menu and save the new script.

Creating a New Script from an Existing Script

To create a new script from an existing script:

1. Click **Open Script** from the **Tools** menu.
2. Select a file from the list of available scripts
3. Click **OK** to enter the Script Editor and display the script.
4. Click **Save Scripts** from the **File** menu and save the new script.

Moving the Insertion Point in a Script

There are two ways to move the insertion point in a script:

- With the mouse
- By specifying a line number

Moving the Insertion Point with the Mouse

Use the following procedure to use the mouse to reposition the insertion point. This approach is especially fast if the area of the screen to which you want to move the insertion point is currently visible.

1. Use the scroll bars at the right and bottom of the display to scroll the target area of the script into view if it is not already visible.
2. Place the mouse pointer where you want to position the insertion point.
3. Click the left mouse button.

   The insertion point is repositioned.

Note: When you scroll the display with the mouse, the insertion point remains in its original position until you reposition it with a mouse click. If you attempt to perform an editing operation when the insertion point is not in view, the Script Editor automatically scrolls the insertion point into view before performing the operation.

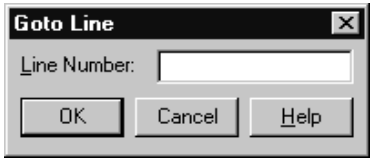
Moving the Insertion Point to a Specified Line in Your Script

Use the following procedure to jump directly to a specified line in your script. This approach is especially fast if the area of the screen to which you want to move the insertion point is not currently visible but you know the number of the target line.

1. Select **Goto Line...** from the **Edit** menu.

   The Script Editor displays the Goto Line dialog.

Figure 6 Goto Line dialog



2. Enter the number of the line in your script to which you want to move the insertion point.
3. Click **OK** button or press ENTER.

4 The insertion point is positioned at the start of the line you specified. If that line was not already displayed, the Script Editor scrolls it into view.

Note: The insertion point cannot be moved so far below the end of a script as to scroll the script entirely off the display. When the last line of your script becomes the first line on your screen, the script will stop scrolling, and you will be unable to move the insertion point below the bottom of that screen.

Selecting Text

There are three ways to select text in an open script:

- With the mouse
- With the keyboard
- By selecting an entire line

Selecting Text with the Mouse

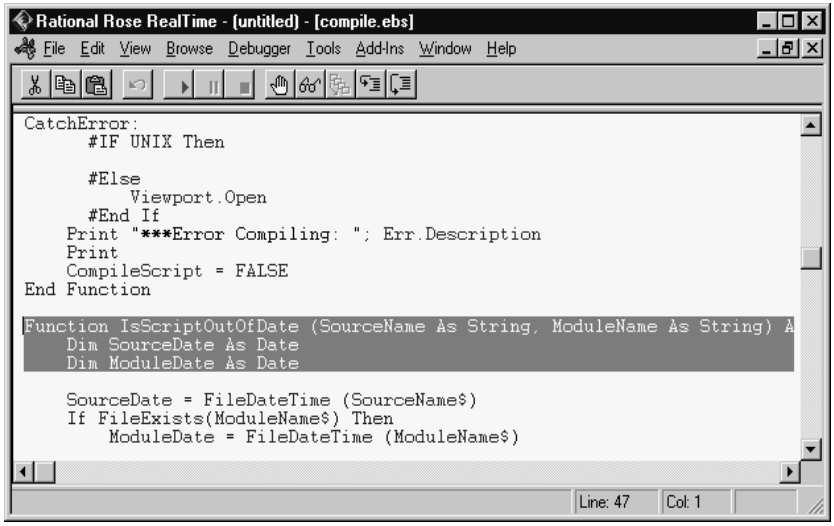
To use the mouse to select text in your script:

1 Place the mouse pointer where you want your selection to begin.

2 Do one of the following:

- While pressing the left mouse button, drag the mouse until you reach the end of your selection, and release the mouse button.
- While pressing SHIFT, place the mouse pointer where you want your selection to end and click the left mouse button.
- The selected text is highlighted on your display.

Figure 7 Selected Scripts Text



Selecting Text with the Keyboard

To use keyboard shortcuts to select text in your script:

1. Place the insertion point where you want your selection to begin.
2. While pressing SHIFT, use one of the navigating keyboard shortcuts to extend the selection to the desired ending point.

   The selected text is highlighted on your display.

Selecting an Entire Line

To use the keyboard to select one or more whole lines in your script:

1. Place the insertion point at the beginning of the line you want to select.
2. Press SHIFT + DOWN ARROW.

   The entire line, including the end-of-line character, is selected.
3. To extend your selection to include additional whole lines of text, repeat step 2.

Deleting, Cutting, Copying, and Pasting Text

Deleting Text

To remove characters, selected text, or entire lines from your script:

- To remove a single character to the left of the insertion point, press BACKSPACE once; to remove a single character to the right of the insertion point, press DELETE once. To remove multiple characters, hold down BACKSPACE or DELETE.
- To remove text that you have selected, press BACKSPACE or DELETE.

Cutting a Selection

To cut text from your script and place it on the Clipboard, press CTRL+X.

Copying a Selection

To copy text from your script and place it on the Clipboard, press CTRL+C.

Pasting the Contents of the Clipboard into Your Script

To paste the contents of the Clipboard into your script:

1. Position the insertion point where you want to place the contents of the Clipboard.
2. Press CTRL+V.

Adding Comments to a Script

There are two types of comments you can add to a script:

- Adding a Full-Line Comment
- Adding a Comment at the End of a Line of Code

Adding a Full-Line Comment

To designate an entire line as a comment:

1. Type an apostrophe (') at the start of the line.
2. Type your comment following the apostrophe.

When your script is run, the presence of the apostrophe at the start of the line will cause the entire line to be ignored.

Adding a Comment at the End of a Line of Code

To designate the last part of a line as a comment:

1. Position the insertion point in the empty space beyond the end of the line of code.
2. Type an apostrophe (').
3. Type your comment following the apostrophe.

When your script is run, the code on the first portion of the line will be executed, but the presence of the apostrophe at the start of the comment will cause the remainder of the line to be ignored.

Finding and Replacing Text

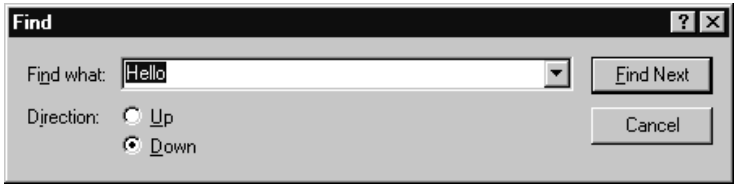
Finding Specified Text

To locate instances of specified text quickly anywhere within your script:

1. Move the insertion point to where you want to start your search. (To start at the beginning of your script, press CTRL+HOME.)
2. Press CTRL+F.

   The Script Editor displays the **Find** dialog:

Figure 8 Find Script Text dialog



3. In the **Find what** field, specify the text you want to find or select it from the list of previous searches.
4. Click **Find Next** or press ENTER.

   The **Find** dialog remains displayed, and the Script Editor either highlights the first instance of the specified text or indicates that it cannot be found.

5 If the specified text has been found, repeat step 4 to search for the next instance of it.

Note: If the *Find* dialog blocks your view of an instance of the specified text, you can move the dialog out of your way and continue with your search. You can also click ***Cancel***, which removes the ***Find*** dialog while maintaining the established search criteria, and then press F3 to find successive occurrences of the specified text.

Replacing Specified Text

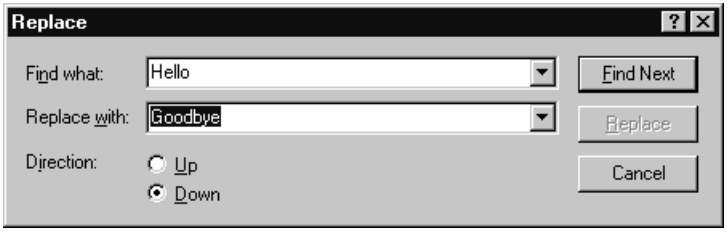
To automatically replace either all instances or selected instances of specified text:

1 Move the insertion point to where you want to start the replacement operation. (To start at the beginning of your script, press CTRL+HOME.)

2 Click **Replace** from the **Edit** menu.

The Script Editor displays the **Replace** dialog:

Figure 9 Replace dialog



3 In the **Find What** field, specify the text you want to replace or select it from the list of previous searches.

4 In the **Replace With** field, specify the replacement text or select it from the list of previous replacements.

5 To replace selected instances of the specified text, click **Find Next**.

The Script Editor either highlights the first instance of the specified text or indicates that it cannot be found.

6 If the specified text has been found, either click **Replace** to replace that instance of it or click **Find Next** to highlight the next instance (if any).

Each time you click **Replace**, the Script Editor replaces that instance of the specified text and automatically highlights the next instance.

Running, Pausing, and Stopping Your Script

Running Your Script

To compile and run your script from within the Script Editor, click **Go** on the toolbar or press F5.

The script is compiled (if it has not already been compiled), the focus is switched to the parent window, and the script is executed.

You can also use the Application Class ExecuteScript method to run scripts. See the *ExecuteScript* method for details.

Pausing an Executing Script

To suspend the execution of a script that you are running, press CTRL+BREAK.

Execution of the script is suspended, and the instruction pointer (a gray highlight) appears on the line of code where the script stopped executing.

Note: The instruction pointer designates the line of code that will be executed next if you resume running your script.

Stopping an Executing Script

Use the following procedure to stop the execution of a script that you are running.

1. If it is not paused, pause the script.
2. Click **StopDebugging** tool on the toolbar (or press SHIFT+F5).

Tracing Script Execution

Stepping Through Your Script

To trace the execution of your script with either the StepInto or StepOver method:

1. Do one of the following:
   - Click the StepInto or StepOver tool on the toolbar.
   - Press F11(StepInto) or F10 (StepOver).

The Script Editor places the instruction pointer on the `sub main` line of your script.

Note: When you initiate execution of your script using either of these methods, the script will first be compiled, if necessary. Therefore, there may be a slight pause before execution actually begins. If your script contains any compile errors, it will not be executed. To debug your script, first correct any compile errors, and then execute it again.

2 To continue tracing the execution of your script, repeat step 1.

3 Each time you repeat step 1, the Script Editor executes the line or the procedure that contains the instruction pointer and then moves the instruction pointer to the next line or procedure to be executed.

4 When you finish tracing the execution of your script, either click **Go** on the toolbar (or press F5) to run the script at full speed or click **Stop Debugging** to halt execution of the script.

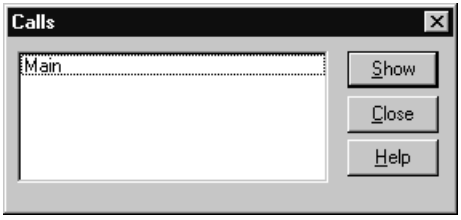
Displaying the Calls dialog

When you are stepping through a subroutine, you may need to determine the procedure calls by which you arrived at that point in your script. Use the following procedure to use the Calls dialog to obtain this information.

1 Click **Calls** on the toolbar.

The Script Editor displays the **Calls** dialog, which lists the procedure calls made by your script in the course of arriving at the present subroutine.

Figure 10 Script Calls dialog



2 From the **Calls** dialog, select the name of the procedure you want to view.

3 Click the **Show** button.

The Script Editor highlights the currently executing line in the procedure you selected, scrolling that line into view if necessary. (During this process, the instruction pointer remains in its original location in the subroutine.)

Setting and Removing Breakpoints

You set and remove breakpoints in your script as part of the debugging process.

Starting Debugging Partway through a Script

To begin the debugging process at a selected point in your script:

1. Place the insertion point in the line where you want to start debugging.
2. To set a breakpoint on that line, click **Toggle Breakpoint** on the toolbar (or press F9).

   The line on which you set the breakpoint now appears in contrasting type.
3. Click **Go** on the toolbar (or press F5).

   The Script Editor runs your script at full speed from the beginning and then pauses prior to executing the line containing the breakpoint. It places the instruction pointer on that line to designate it as the line that will be executed next when you either proceed with debugging or resume running the script.

Continuing Debugging at a Line Outside the Current Subroutine

To continue debugging at a line that *isn't* within the same subroutine, use the following procedure to move the instruction pointer to that line.

1. Place the insertion point in the line where you want to continue debugging.
2. To set a breakpoint on that line, press F9.
3. To run your script, click **Go** on the toolbar (or press F5).

   The script executes at full speed until it reaches the line containing the breakpoint and then pauses with the instruction pointer on that line. You can now resume stepping through your script from that point.

Debugging Selected Portions of Your Script

To debug parts of your script, use the following procedure to facilitate the task by using breakpoints.

1. Place a breakpoint at the start of each portion of your script that you want to debug.

Note: Up to 255 lines in your script can contain breakpoints.

2. To run the script, click **Go** on the toolbar or press F5.

   The script executes at full speed until it reaches the line containing the first breakpoint and then pauses with the instruction pointer on that line.

3 Step through as much of the code as you need to.

4 To resume running your script, click **Go** on the toolbar or press F5.

The script executes at full speed until it reaches the line containing the second breakpoint and then pauses with the instruction pointer on that line.

5 Repeat steps 3 and 4 until you have finished debugging the selected portions of your script.

Removing a Single Breakpoint Manually

To delete breakpoints manually one at a time:

1 Place the insertion point on the line containing the breakpoint that you want to remove.

2 Click **Toggle Breakpoint** on the toolbar, or press F9.

The breakpoint is removed, and the line no longer appears in contrasting type.

Removing All Breakpoints Manually

To delete all breakpoints manually in a single operation, click **Clear All Breakpoints** from the **Debugger** menu.

Working with Watch Variables

Watch variables allow you to track the changing values of variables in a script.

Adding Watch Variables

To add a variable to the Script Editor's watch variable list:

1 Click **Add Watch** on the toolbar or press CTRL+F9.

The Script Editor displays the **Add Watch** dialog.

Figure 11 Add Watch dialog

2 Use the controls in the Context box to specify where the variable is defined (locally, publicly, or privately) and, if it is defined locally, in which routine it is defined.

3 In the **Variable Name** field, enter the name of the variable you want to add to the watch variable list.

You can only watch variables of fundamental data types, such as Integer, Long, Variant, and so on; you cannot watch complex variables such as structures or arrays. You can, however, watch individual elements of arrays or structure members.

Use the following syntax to watch individual elements of arrays or structure members in a script:

```
[variable [(index,...)] [.member [(index,...)]]...]
```

Where **`variable`** is the name of the structure or array variable, **`index`** is a literal number, and **`member`** is the name of a structure member.

For example, the following are valid watch expressions:

Table 1 Sample Watch Expressions

Watch Variable	Description
a(1)	Element 1 of array **a**
person.age	Member **age** of structure **person**
company(10,23).person.age	Member **age** of structure **person** that is at element 10,23 within the array of structures called **company**

Note: If you are executing the script, you can display the names of all the variables that are "in scope," or defined within the current function or subroutine, on the drop-down Variable Name list and select the variable you want from that list.

4 Click **OK** or press ENTER.

If this is the first variable you are placing on the watch variable list, the watch pane opens far enough to display that variable. If the watch pane was already open, it expands far enough to display the variable you just added.

Note: Although you can add as many watch variables to the list as you want, the watch pane only expands until it fills half of the Script Editor's application window. If your list of watch variables becomes longer than that, you can use the watch pane's scroll bars to bring hidden portions of the list into view.

Selecting Variables on the Watch List

In order to delete a variable from the Script Editor's watch variable list or modify the value of a variable on the list, do one of the following:

- Place the mouse pointer on the variable you want to select and click the left mouse button.
- If one of the variables on the watch list is already selected, use the arrow keys to move the selection highlight to the desired variable.
- If the insertion point is in the edit pane, press F6 to highlight the most recently selected variable on the watch list and then use the arrow keys to move the selection highlight to the desired variable.

  Note: Pressing F6 again returns the insertion point to its previous position in the edit pane.

Deleting Watch Variables

To delete a selected variable from the Script Editor's watch variable list:

1. Select the variable on the watch list.
2. Click **Delete Watch** from the **Debugger** menu, or press DELETE.

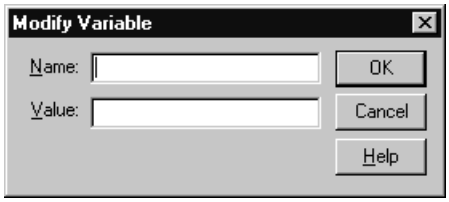
Modifying the Value of Variables on the Watch Variable List

When the debugger has control, you can modify the value of any of the variables on the Script Editor's watch variable list. Use the following procedure to change the value of a selected watch variable.

1. Place the mouse pointer on the name of the variable whose value you want to modify and double-click the left mouse button.
2. Select the name of the variable whose value you want to modify and press ENTER or F2.

The Script Editor displays the Modify Variable dialog.

Figure 12 Modify Variable dialog



Note: The name of the variable you selected on the watch variable list appears in the ***Name*** field.

When you use the **Modify Variable** dialog to change the value of a variable, you don't have to specify the context. The Script Editor first searches locally for the definition of that variable, then privately, then publicly.

3 Enter the new value for your variable in the **Value** field.

4 Click the **OK** button.

The new value of your variable appears on the watch variable list.

Compiling Your Script

To create compiled script files from your script source:

1 Click **Open Script** from the **Tools** menu and select the file that contains the script you want to compile.

2 Click **Compile** from the **Debugger** menu, or press F7.

3 Enter the name of the file in which to save the compiled script and select **OK**.

The script is compiled and saved in a file with a .ebx extension.

Note: You can also use the Application.CompileScriptFile method to compile scripts. Check the Extensibility Reference or the Extensibility Online Help for more details.

Using Interscript Calls

Guidelines for Using a Script to Call Another Script

You can write a script that includes code that calls and executes another script. The following guidelines apply to this process:

- You can only call and execute a compiled script from within another script.
- Use the LoadScript method to load the script into memory.
- Use the FreeScript to unload the script from memory.
- Even if you call LoadScript multiple times, the script is only loaded into memory one time. However, for each LoadScript call you make, you must include a corresponding FreeScript call. If you do not do this, the script will not be unloaded from memory.

Debugging Interscript Calls

To debug a script that uses interscript calls:

1. Enter the call to the compiled script you are including and set a breakpoint on the call.
2. Click **StepInto** from the **Debugger** menu.

The Script Editor displays the source code for the compiled script you are calling, and steps through it line by line.

When the trace of the called script is complete, the Script Editor redisplays the calling script.

Note: The script you are calling must be compiled with debugging turned on. See Compiling Your Script, earlier in this chapter, for details.

Working with the Dialog Editor

Inserting a Dialog into Your Script

To insert a dialog into your script:

1. Place the insertion point where you want the BasicScript code for the dialog to appear in your script.
2. From the **Edit** menu, click **Insert Dialog**.

The Script Editor's application window is temporarily disabled, and Dialog Editor appears, displaying a new dialog in its application window.

3 Use the Dialog Editor to create your dialog.

4 Exit and Return from Dialog Editor and return to the Script Editor.

The Script Editor automatically places the code for the dialog in your script at the location of the insertion point.

Editing an Existing Dialog

To edit an existing dialog template in your script:

1 Select the BasicScript code for the entire dialog template.

2 From the **Edit** menu, click **Edit Dialog**.

The Script Editor's application window is temporarily disabled, and Dialog Editor appears, displaying in its application window a dialog created from the code you selected.

3 Use the Dialog Editor to modify your dialog.

4 Exit from the Dialog Editor and return to the Script Editor.

The Script Editor automatically replaces the BasicScript code you originally selected with the revised code generated by the Dialog Editor.

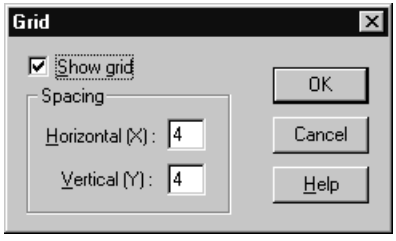
Displaying and Adjusting the Grid

To display and adjust the X and Y settings, which can help you position controls more precisely within your dialog:

1 Press CTRL+G.

The Dialog Editor displays the following dialog:

Figure 13 Grid Dialog



2 To display the grid in your dialog, click **Show grid**.

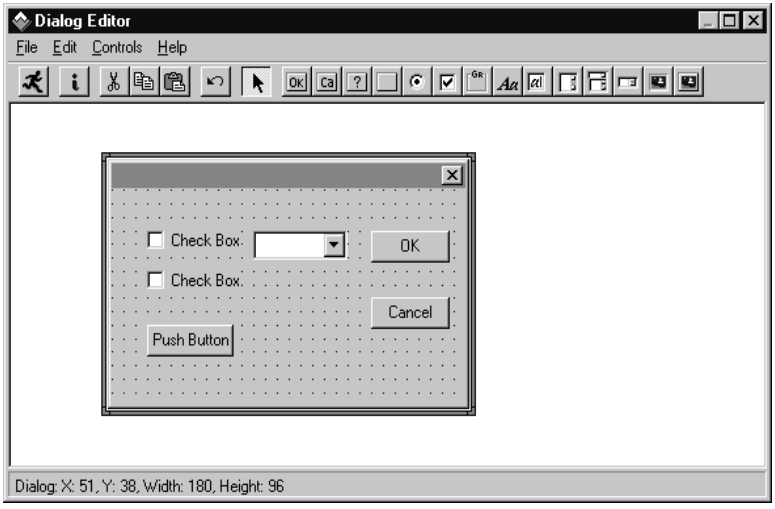
3 To change the current X and Y settings, enter new values in the X and Y fields.

Note: The values of X and Y in the Grid dialog determine the grid's spacing. Assigning smaller X and Y values produces a more closely spaced grid, which enables you to move the mouse pointer in smaller horizontal and vertical increments as you position controls. Assigning larger X and Y values produces the opposite effect on both the grid's spacing and the movement of the mouse pointer. The X and Y settings entered in the Grid dialog remain in effect regardless of whether you choose to display the grid.

4 Click **OK** or press ENTER.

The Dialog Editor displays the grid with the settings you specified.

Figure 14 Dialog Edition with Grid Displayed



5 With the grid displayed, line up the crosshairs on the mouse pointer with the dots on the grid to position controls precisely and align them with respect to other controls.

Changing Titles and Labels

Use the following procedure to change the title of a dialog, as well as the labels of group boxes, option buttons, push buttons, text controls, and check boxes:

1. Display the **Information** dialog for the dialog whose title you want to change or for the control whose label you want to change.
2. Enter the new title or label in the **Text$** field.

Note: Dialog titles and control labels are optional. Therefore, you can leave the ***Text$*** field blank.

3. If the information in the **Text$** field should be interpreted as a variable name rather than a literal string, click **Variable Name**.
4. Click **OK** or press ENTER.

   The new title or label is now displayed on the title bar or on the control.

Assigning Accelerator Keys

To designate a letter from a control's label to serve as the accelerator key for that control:

1. Display the **Information** dialog for the control to which you want to assign an accelerator key.
2. In the **Text$** field, type an ampersand (&) before the letter you want to designate as the accelerator key.
3. Click **OK** or press ENTER.

The letter you designated is now underlined on the control's label, and users will be able to access the control by pressing ALT + the underlined letter.

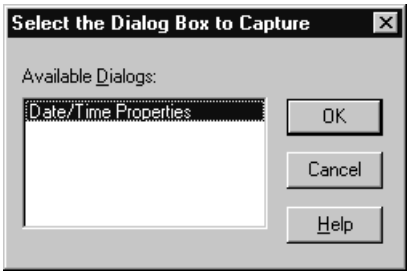
Capturing Standard Windows Dialogs

Use the following procedure to capture the standard Windows controls from any standard Windows dialog in another application, and insert those controls into the Dialog Editor for editing:

1. Display the dialog you want to capture.
2. Open the Dialog Editor.
3. Click **Capture Dialog** from the **File** menu.

The Dialog Editor displays a dialog that lists all open dialogs that it is able to capture:

Figure 15 Capturing a Dialog



4 Select the dialog that you want to capture, then click **OK**.

Note: The Dialog Editor only supports standard Windows controls and standard Windows dialogs. Therefore, if the target dialog contains both standard Windows controls and custom controls, only the standard Windows controls will appear in the Dialog Editor's application window. If the target dialog is not a standard Windows dialog, you will be unable to capture the dialog or any of its controls.

Testing Your Dialogs

The Dialog Editor lets you run your edited dialog purposes. When you click **Test**, your dialog comes alive, which gives you an opportunity to make sure it functions properly and fix any problems before you incorporate the dialog template into your script.

Before you run your dialog, take a moment to look it over for basic problems such as the following:

- Does the dialog contain a command button - that is, a default **OK** or **Cancel** button, a push button, or a picture button?
- Does the dialog contain all the necessary push buttons?
- Does the dialog contain a Help button if one is needed?
- Are the controls aligned and sized properly?
- If there is a text control, is its font set properly?
- Are the close box and title bar displayed (or hidden) as you intended?
- Are the control labels and dialog title spelled and capitalized correctly?
- Do all the controls fit within the borders of the dialog?

- Could you improve the design of the dialog by adding one or more group boxes to set off groups of related controls?
- Could you clarify the purpose of any unlabeled control (such as a text box, list box, combo box, drop list box, picture, or picture button) by adding a text control to serve as a de facto label for it?
- Have you made all the necessary accelerator key assignments?
- After you've fixed any elementary problems, you're ready to run your dialog so you can check for problems that don't become apparent until a dialog is activated.

Testing your dialog is an iterative process that involves running the dialog to see how well it works, identifying problems, stopping the test and fixing those problems, then running the dialog again to make sure the problems are fixed and to identify any additional problems, and so forth—until the dialog functions the way you intend.

To test your dialog and fine-tune its performance:

1. Click **Run** on the toolbar, or press F5, to make the dialog operational.
2. Check the dialog's functions.
3. To stop the test, click **Run**, press F5, or double-click the dialog's close box (if it has one).
4. Make any necessary adjustments to the dialog.
5. Repeat steps 1-4 as many times as you need in order to get the dialog working properly.

Incorporating Dialogs or Controls into Your Script

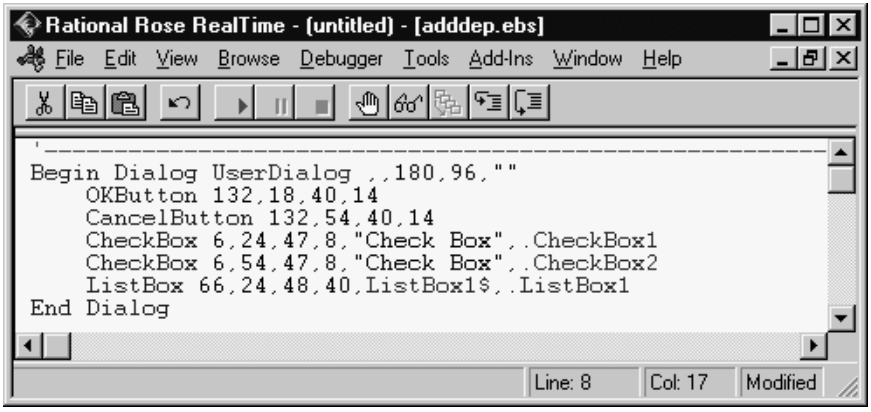
You create dialogs and dialog controls in the Dialog Editor. To incorporate them into a script, you copy them to the Clipboard. When you copy the dialog to the Clipboard, it is stored in the form of Basic Script statements. You then paste the contents of the Clipboard into the script.

To incorporate a dialog or control into your script:

1. Select the dialog or control that you want to incorporate into your script.
2. Press CTRL+C.
3. Open your script and paste in the contents of the Clipboard at the desired point.

   The dialog template or control is now described in BasicScript statements in your script, as shown in the following example

Figure 16 Sample Dialog in Basic Script



Selecting Controls

Do one of the following to select a control in a dialog:

- With the **Pick** tool active, place the mouse pointer on the desired control and click the mouse button.
- With the **Pick** tool active, press the TAB key repeatedly until the focus moves to the desired control.

  The control is now surrounded by a thick frame to indicate that it is selected and you can edit it.

Selecting Dialogs

Do one of the following to select an entire dialog:

- With the **Pick** tool active, place the mouse pointer on the title bar of the dialog or on an empty area within the borders of the dialog (that is, on an area where there are no controls) and click the mouse button.
- With the **Pick** tool active, press the TAB key repeatedly until the focus moves to the dialog.

The dialog is now surrounded by a thick frame to indicate that it is selected and you can edit it.

Repositioning Items

Repositioning Items with the Mouse

To reposition items in a dialog or control by dragging it with the mouse:

1. With the **Pick** tool active, place the mouse pointer on an empty area of the dialog or on a control.
2. Depress the mouse button and drag the dialog or control to the desired location.

Note: The increments by which you can move a control with the mouse are governed by the grid setting. For example, if the grid's X setting is 4 and its Y setting is 6, you'll be able to move the control horizontally only in increments of 4 X units and vertically only in increments of 6 Y units. This feature is handy if you're trying to align controls in your dialog. If you want to move controls in smaller or larger increments, press CTRL+G to display the Grid dialog and adjust the X and Y settings.

Repositioning Items with the Arrow Keys

To reposition items in a dialog or control by dragging it with the arrow keys:

1. Select the dialog or control that you want to move.
2. Do one of the following:
   - Press an arrow key once to move the item by 1 X or Y unit in the desired direction.
   - Steadily press an arrow key to “nudge” the item gradually along in the desired direction.

Note: When you reposition an item with the arrow keys, a faint, partial afterimage of the item may remain visible in the item's original position. These afterimages are rare and will disappear once you test your dialog.

Repositioning Dialogs with the Dialog Information Dialog

Use the following procedure to reposition items in a dialog or control by using the Dialog Information dialog.

1. Display the **Information** dialog.

   Note: For information on displaying the Dialog Information dialog, see Displaying the Dialog Information dialog, later in this chapter.

2 Do one of the following:

- Change the X and Y coordinates in the **Position** group box.
- Leave the X and/or Y coordinates blank.

3 Click **OK** or press ENTER.

If you specified X and Y coordinates, the dialog moves to that position. If you left the X coordinate blank, the dialog will be centered horizontally relative to the parent window of the dialog when the dialog is run. If you left the Y coordinate blank, the dialog will be centered vertically relative to the parent window of the dialog when the dialog is run.

Repositioning Controls with the Dialog Information Dialog

1 Use the following procedure to move a selected control by changing its coordinates in the **Dialog Information** dialog for that control.

Note: For information on displaying the Dialog Information dialog, see Displaying the Dialog Information dialog, later in this chapter.

2 Display the **Information** dialog for the control that you want to move.

3 Change the X and Y coordinates in the **Position** group box.

4 Click **OK** or press ENTER.

The control moves to the specified position.

Resizing Items

Resizing Items with the Mouse

To change the size of a selected dialog or control by dragging its borders or corners with the mouse:

1 With the **Pick** tool active, select the dialog or control that you want to resize.

2 Place the mouse pointer over a border or corner of the item.

3 Depress the mouse button and drag the border or corner until the item reaches the desired size.

Resizing Items with the Information Dialog

To change the size of a selected dialog or control by changing its **Width** or **Height** settings in the **Information** dialog.

1. Display the **Information** dialog for the dialog or control that you want to resize.
2. Change the **Width** and **Height** settings in the **Size** group box.
3. Click the **OK** button or press ENTER.

   The dialog or control is resized to the dimensions you specified.

Resizing Selected Items Automatically

You can adjust the borders of certain controls automatically to fit the text displayed on them.

To resize selected controls automatically:

1. With the **Pick** tool active, select the option button, text control, push button, check box, or text box that you want to resize.
2. Press F2.

   The borders of the control will expand or contract to fit the text displayed on it.

Adding Controls

Use the following procedure to add one or more controls to your dialog using simple mouse and keyboard methods.

1. From the toolbar, choose the tool corresponding to the type of control you want to add.

   Note: When you pass the mouse pointer over an area of the display where a control can be placed, the pointer becomes an image of the selected control with crosshairs (for positioning purposes) to its upper left. The name and position of the selected control appear on the status bar. When you pass the pointer over an area of the display where a control cannot be placed, the pointer changes into a circle with a slash through it (the “prohibited” symbol).

   Note: You can only insert a control within the borders of the dialog you are creating. You cannot insert a control on the dialog's title bar or outside its borders.
2. Place the pointer where you want the control to be positioned and click the mouse button.

The control you just created appears at the specified location. (To be more specific, the upper left corner of the control will correspond to the position of the pointer's crosshairs at the moment you clicked the mouse button.) The control is surrounded by a thick frame, which means that it is selected, and it may also have a default label.

After the new control has appeared, the mouse pointer becomes an arrow, to indicate that the Pick tool is active and you can once again select any of the controls in your dialog.

3 To add another control of the same type as the one you just added, press CTRL+D.

A duplicate copy of the control appears.

4 To add a different type of control, repeat steps 1 and 2.

5 To reactivate the Pick tool, do one of the following:

- Click the arrow-shaped tool on the toolbar.
- Place the mouse pointer on the title bar of the dialog or outside the borders of the dialog (that is, on any area where the mouse pointer turns into the "prohibited" symbol) and click the mouse button.

Duplicating Controls

Use the following procedure to use the Dialog Editor's duplicating feature, which saves you the work of creating additional controls individually if you need one or more copies of a particular control:

1 Select the control that you want to duplicate.

2 Press CTRL+D.

A duplicate copy of the selected control appears in your dialog.

3 Repeat step 2 as many times as necessary to create the desired number of duplicate controls.

Adding Pictures to a Dialog

You can add pictures to a dialog from a file or from a picture library.

Adding Pictures from Files

Use the following procedure to display a Windows bitmap or metafile from a file on a picture control or picture button control by using the control's Information dialog to indicate the file in which the picture is contained.

1. Display the **Information** dialog for the picture control or picture button control whose picture you want to specify.
2. In the **Picture source** option button group, click **File**.
3. In the **Name$** field, enter the name of the file containing the picture you want to display in the picture control or picture button control.

Note: By clicking the ***Browse*** button, you can display the ***Select a Picture File*** dialog and use it to find the file.

4. Click **OK** or press ENTER.

   The picture control or picture button control now displays the picture you specified.

Adding Pictures from Picture Libraries

Use the following procedure to display a Windows bitmap or metafile from a file on a picture control or picture button control by using the control's **Information** dialog to indicate the file in which the picture is contained.

1. Display the **Information** dialog for the picture control or picture button control whose picture you want to specify.
2. In the **Picture source** option button group, click **File**.
3. In the **Name$** field, enter the name of the file containing the picture you want to display in the picture control or picture button control.

Note: By clicking the ***Browse*** button, you can display the ***Select a Picture File*** dialog and use it to find the file.

4. Click **OK** or press ENTER.

   The picture control or picture button control now displays the picture you specified.

Pasting Items into Dialog Editor

Pasting Existing Dialogs into Dialog Editor

If you want to modify a BasicScript dialog template contained in your script, use the following procedure to select the template and paste it into dialog editor for editing:

1. Copy the entire **BasicScript** dialog template (from the `Begin Dialog` instruction to the `End Dialog` instruction) from your script to the Clipboard.
2. Open the Dialog Editor.
3. Press CTRL+V.
4. When the Dialog Editor asks whether you want to replace the existing dialog, click **Yes**.

   The Dialog Editor creates a new dialog corresponding to the template contained on the Clipboard.

Pasting Controls from Existing Dialogs into Dialog Editor

If you want to modify the BasicScript statements in your script that correspond to one or more dialog controls, use the following procedure to select the statements and paste them into Dialog Editor for editing:

1. Copy the BasicScript description of the control(s) from your script to the Clipboard.
2. Open Dialog Editor.
3. Press CTRL+V.

   Dialog Editor adds to your current dialog one or more controls corresponding to the description contained on the Clipboard.

Displaying the Information Dialogs

There are two types of Information dialogs:

- Information dialog for Dialogs
- Information dialog for Controls

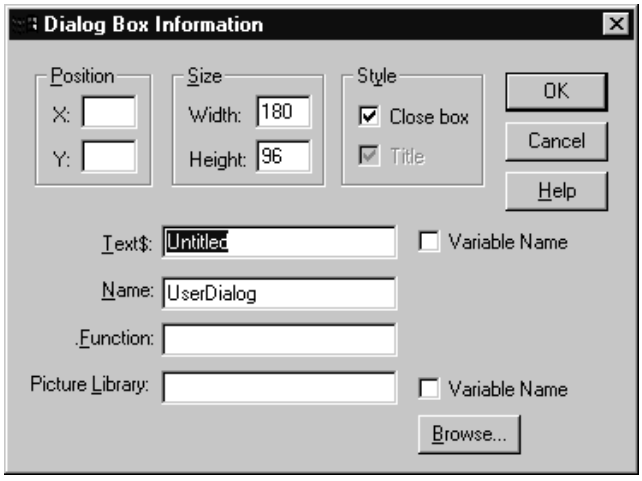
Displaying the Information Dialogs for Dialogs

Do one of the following to display the **Information** dialog to check and adjust attributes that pertain to the dialog as a whole:

- With the Pick tool active, place the mouse pointer on an area of the dialog where there are no controls and double-click the mouse button.
- With the Pick tool active, select the dialog and either click the Information tool on the toolbar, press ENTER, or press CTRL+I.

  The following figure shows the dialog Information dialog:

Figure 17 Dialog Information Dialog



Attributes You Can Adjust with the Dialog Information dialog

The dialog Information dialog can be used to check and adjust the following attributes, which pertain to the dialog as a whole.

- **Position** (optional): X and Y coordinates on the display, in dialog units
- **Size** (mandatory): width and height of the dialog, in dialog units
- **Style** (optional): options that allow you to determine whether the close box and title bar are displayed
- **Text$** (optional): text displayed on the title bar of the dialog

- **Name** (mandatory): name by which you refer to this dialog template in your BasicScript code
- **Function** (optional): name of a BasicScript function in your dialog
- **Picture Library** (optional): picture library from which one or more pictures in the dialog are obtained

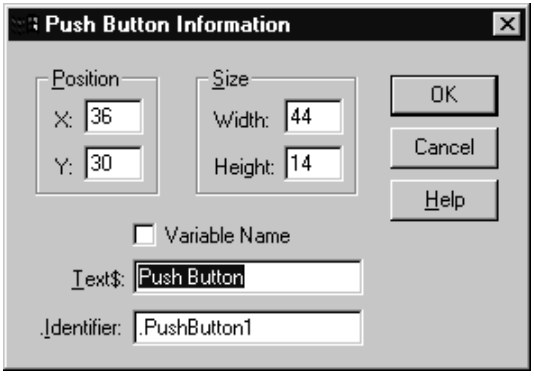
Displaying the Information Dialogs for Controls

Do one of the following to display the Information dialog for a control to check and adjust attributes that pertain to that particular control.

- With the **Pick** tool active, place the mouse pointer on the desired control and double-click the mouse button.
- With the **Pick** tool active, select the control and either click the Information tool on the toolbar, press ENTER, or press CTRL+I.

  The Dialog Editor displays an Information dialog corresponding to the control you selected. For example:

Figure 18 Control Information dialog



Attributes You Can Adjust with the Information Dialogs for Controls

Control Information dialogs can be used to check and adjust the attributes of the following controls:

- Default OK Button Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
  - Default Cancel Button Information dialog
    - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
    - **Size** (mandatory): width and height of the control, in dialog units.
    - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
- Help Button Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **FileName$** (optional): Name of the help file that you want to invoke.
  - **Context&** (mandatory): The context ID specifying which help topic to jump to.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
- Push Button Information dialog.
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Text$** (optional): text displayed on a control.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.

- Option Button Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Text$** (optional): text displayed on a control.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
  - **Option Group** (mandatory): name by which you refer to a group of option buttons in your BasicScript code.
- Check Box Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Text$** (optional): text displayed on a control.
  - **Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog has been processed.
- Group Box Information dialog.
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Text$** (optional): text displayed on a control.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
- Text Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Text$** (optional): text displayed on a control.
  - **Font** (optional): font in which text is displayed.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.

- Text Box Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Multiline** (optional): option that allows you to determine whether users can enter a single line of text or multiple lines.
  - **Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog has been processed.
- List Box Information dialog.
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog has been processed.
  - **Array$** (mandatory): name of an array variable in your BasicScript code.
- Combo Box Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog has been processed.
  - **Array$** (mandatory): name of an array variable in your BasicScript code.
- Drop List Box Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Identifier** (mandatory): name by which you refer to a control in your BasicScript code; also contains the result of the control after the dialog has been processed.
  - **Array$** (mandatory): name of an array variable in your BasicScript code.

- Picture Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units.
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
  - **Identifier** (optional): name of the file containing a picture that you want to display or the name of a picture that you want to display from a specified picture library.
  - **Frame** (optional): option that allows you to display a 3-D frame.
- Picture Button Information dialog
  - **Position** (mandatory): X and Y coordinates within the dialog, in dialog units.
  - **Size** (mandatory): width and height of the control, in dialog units
  - **Identifier** (optional): name by which you refer to a control in your BasicScript code.
  - **Identifier** (optional): name of the file containing a picture that you want to display or the name of a picture that you want to display from a specified picture library.

Opening a Model

OpenModel will always return a model. If **OpenModel** fails to open the specified model, it will return the default empty model. To verify that the model you wanted was opened, use code similar to the following:

```
Set theModel = theApplication.OpenModel(modelFileName)
If Not theModel.GetFileName () = modelFileName Then
    theApplication.WriteErrorLog "Bad model"
End If
```

Modifying a Property Value

A property is a user-extensible part of the RRTEI that allows name-value pairs to be attached to every model element. Properties capture information that is specific to a particular project or add-in.

We will use RRTEI to modify a value for a protocol. For example, the **TypeSafeSignals** property on the C++ TargetRTS tab for a Protocol specification property (C++ Language Add-In). The Property class has the following public attributes (see the Extensibility Interface Reference for Rational Rose RealTime):

Name : String	**Name of the property**
ToolName : String	A tool can be a programming language tool (such as C++) or a user-defined Add-in to Rational Rose RealTime. A tool corresponds to a tab in the property specification; however the ToolName and the tab title are not always identical.
Type : String	Indicates the type of information stored by the property.
Value : String	Indicates the value of the property

The **OverrideProperty** operation allows you to modify the value of a property for a particular element:

```
OverrideProperty (theToolName : String, thePropName : String,
theValue : String) : Boolean
```

To use this function, you need to know the tool name, which does not necessarily match the title on the tab (for example, C++ TargetRTS). To find this information, we can create a Rose RealTime script that queries a protocol element to find its properties and their associated tool names. The following subroutine takes a protocol element and prints all of its properties:

```
Sub PrintProperties (theProtocol As RoseRT.Protocol)
      Dim allProperties As RoseRT.PropertyCollection
      Dim theProperty As RoseRT.Property
      Set allProperties = theProtocol.GetAllProperties()
      For i = 1 To allProperties.Count
              Set theProperty = allProperties.GetAt(i)
              Print "Name: "; theProperty.Name
              Print Spc(5); "Value: "; theProperty.Value
              Print Spc(5); "ToolName: "; theProperty.ToolName
```

```
            Print Spc(5); "Type: "; theProperty.Type
      Next i
End Sub
```

The output looks similar to the following:

```
Name: BackwardsCompatible
        Value: False
        ToolName: OT::CppTargetRTS
        Type: Boolean
Name: Version
        Value: 0
        ToolName: OT::CppTargetRTS
        Type: Integer
Name: TypeSafeSignals
        Value: True
        ToolName: OT::CppTargetRTS
        Type: Boolean
```

You now have all the information required to use the OverrideProperties function.

Note: It is important to use caution when using the OverrideProperties function; specifying a property name that does not exist causes the creation of a new property instead of modifying an existing one.

The following subroutine de-selects (un-checks) the TypeSafeSignals box for the specified protocol:

```
Sub TurnOffTypeSafeSignals (theProtocol As RoseRT.Protocol)
      If theProtocol.IsModifiable Then
            Print "Changing properties of: "; theProtocol.Name
            If Not theProtocol.OverrideProperty("OT::CppTargetRTS",
            "TypeSafeSignals", "False") Then
                  Print "Error modifying the properties of protocol:
";
theProtocol.Name
            End If
      End If
End Sub
```

Note: The **IsModifiable** function call is necessary to verify that the model element can be modified (for example, it was checked out, if necessary, and not read-only).

This example illustrates how to modify the TypeSafeSignals property for a protocol defined by the C++ Language Add-In. However, you can create subroutines to modify any of the properties available for Rose RealTime Add-Ins. The properties are not always documented in the online help, but you can use the GetAllProperties function to determine the name, type, and associated tool for all properties.

For additional information, contact Rational Customer Support.

Setting the Top Capsule of a Component

The TopCapsule field for a component is a property of the specific Language Add-In used. This property is a structured property which is not thoroughly documented in the online help.

The following subroutine sets the TopCapsule field for a component given the component and the capsule:

```
Sub SetTopCapsule (theComponent As RoseRT.Component, theCapsule As
RoseRT.Capsule)
      ' First add the capsule as a reference if it isn't already
      If theComponent.AssignedClasses.FindFirst(theCapsule.Name) = 0
Then
              If Not theComponent.AssignClass(theCapsule) Then
                      MsgBox "Error configuring component."
                      Exit Sub
              End If
      End If
toolName$ = "OT::CppExec"         'Modify this for other Language
Add-Ins
      propertyName$ = "TopCapsule"
' If you print out the "TopCapsule"property it looks like this:
      ' [event_ui
      ' description='MyCapsule'
      ' caption='Select...']
      ' "Logical View::MyCapsule" 39B53F390336
      ' This is a structured property, that is, it contains sections
     ' (e.g. event_ui) that contain field names (e.g. description) and
      ' values (for example, MyCapsule).   It also contains the model
path
      ' and unique id entry.
```

```
	' Since there is no OverrideProperty function that takes a
	' StructuredProperty, we first have to override the default
property,
	' get its StructuredPropery, and modify this.
	If Not theComponent.OverrideProperty(toolName, propertyName, "")
Then
		MsgBox "Error configuring component."
		Exit Sub
	End If

	Dim sp As RoseRT.StructuredProperty
	Set sp =
theComponent.GetToolProperties(toolName).GetFirst(propertyName)
	sp.SetFieldValue "event_ui", "description", theCapsule.Name
	sp.SetFieldValue "event_ui", "caption", "Select..."

	Dim fullCapsuleName As String
	fullCapsuleName = """" + theCapsule.GetQualifiedName() + """" + "
" +  theCapsule.GetUniqueID()
	sp.SetFieldValue "", "", fullCapsuleName
End Sub
```

The following script illustrates how to use the **SetTopCapsule** subroutine (described above). This script creates components for all the capsules in the model, and puts them in to a component package called **ComponentsForAll**.

```
Dim theModel As RoseRT.Model

Sub SetTopCapsule (theComponent As RoseRT.Component, theCapsule As
RoseRT.Capsule)
	' First add the capsule as a reference if it isn't already
	If theComponent.AssignedClasses.FindFirst(theCapsule.Name) = 0
Then
		If Not theComponent.AssignClass(theCapsule) Then
			MsgBox "Error configuring component."
			Exit Sub
		End If
	End If
```

```
     toolName$ = "OT::CppExec"        'Modify this for other Language
Add-Ins
      propertyName$ = "TopCapsule"

      ' If you print out the "TopCapsule"property it looks like this:
      ' [event_ui
      ' description='MyCapsule'
      ' caption='Select...']
      ' "Logical View::MyCapsule" 39B53F390336
      ' This is a structured property, that is, it contains sections
     ' (e.g. event_ui) that contain field names (e.g. description) and
      ' values (e.g. MyCapsule).   It also contains the model path
      ' and unique id entry.
      ' Since there is no OverrideProperty function that takes a
      ' StructuredProperty, we first have to override the default
property,
      ' get its StructuredPropery, and modify this.
     If Not theComponent.OverrideProperty(toolName, propertyName, "")
Then
              MsgBox "Error configuring component."
              Exit Sub
      End If

      Dim sp As RoseRT.StructuredProperty
      Set sp =
theComponent.GetToolProperties(toolName).GetFirst(propertyName)
      sp.SetFieldValue "event_ui", "description", theCapsule.Name
      sp.SetFieldValue "event_ui", "caption", "Select..."

      Dim fullCapsuleName As String
     fullCapsuleName = """" + theCapsule.GetQualifiedName() + """" + "
" +
theCapsule.GetUniqueID()
      sp.SetFieldValue "", "", fullCapsuleName
End Sub


Sub myCreateComponent ( thisCapsule As RoseRT.Capsule )
```

```
	' local strings
	ComponentsForAll$ = "ComponentsForAll"

	' for retrieving the component
	Dim myComponent As RoseRT.Component
	Dim myComponents As RoseRT.ComponentCollection
	Dim theComponentPackages As RoseRT.ComponentPackageCollection
	Dim myComponentPackage As RoseRT.ComponentPackage

	' set up Package for Components created with script if it does
not exist.
	Set theComponentPackages =
theModel.RootComponentPackage.GetAllComponentPack
ages( )
	i = theComponentPackages.FindFirst(ComponentsForAll)
	If i = 0 Then
		Set myComponentPackage =
theModel.RootComponentPackage.AddComponentPacka
ge ( ComponentsForAll )
	Else
		Set myComponentPackage = theComponentPackages.GetAt(i)
	End If

	' add component if it does not already exist
	Set myComponents = myComponentPackage.GetAllComponents( )
	i = myComponents.FindFirst(thisCapsule.Name)
	If i = 0 Then
		Set myComponent =
myComponentPackage.AddComponent(thisCapsule.Name)
		SetTopCapsule myComponent, thisCapsule
	End If
End Sub

Sub Main
	Dim theCapsules As RoseRT.CapsuleCollection
	Dim myCapsule As RoseRT.Capsule
```

```
        Set theModel = RoseRTApp.CurrentModel

        ' retrieve the capsules
        Set theCapsules = theModel.GetAllCapsules ( )
        For i = 1 To theCapsules.Count
                Set myCapsule = theCapsules.GetAt (i)
                ' the next if statement is to avoid creating
                ' components that reference capsules not owned
                ' by the Model (i.e. in RTClasses)
                If   myCapsule.isOwned Then
                        myCreateComponent myCapsule
                End If
        Next i
End Sub
```

For additional information, contact Rational Customer Support.

Rational Rose RealTime Extensibility Interface Reference

3

Contents

This chapter is organized as follows:

- *Logical Package Structure* on page 80
- *Application Classes* on page 81
- *AddIn* on page 86
- *Application* on page 93
- *ContextMenuItem* on page 122
- *MenuState* on page 123
- *PathMap* on page 124
- *RsMenuState* on page 127
- *Workspace* on page 128
- *Extensibility Classes* on page 130
- *Collection* on page 131
- *RoseBase* on page 139
- *RRTEIObject* on page 140
- *RichTypes* on page 141
- *RichType* on page 142
- *RichTypeValuesCollection* on page 144
- *Model Classes* on page 145
- *Component View Classes* on page 145
- *Component* on page 149
- *ComponentPackage* on page 170
- *Core Model Classes* on page 178
- *ControllableElement* on page 184
- *DefaultModelProperties* on page 194
- *Element* on page 204
- *ExternalDocument* on page 215
- *Model* on page 218
- *ModelElement* on page 236
- *Package* on page 239
- *Property* on page 243
- *RsExternalDocumentType* on page 244
- *StructuredProperty* on page 244

- *Deployment View Classes* on page 246
- *ComponentInstance* on page 249
- *DeploymentPackage* on page 252
- *Device* on page 258
- *Processor* on page 262
- *Logical View Classes* on page 267
- *LogicalPackage* on page 269
- *Association Classes* on page 288
- *Association* on page 290
- *AssociationEnd* on page 294
- *AssociationEndContainment* on page 298
- *AssociationEndVisibilityKind* on page 299
- *Classifier Classes* on page 299
- *Capsule* on page 303
- *Class* on page 304
- *ClassConcurrency* on page 310
- *ClassKind* on page 310
- *Classifier* on page 310
- *ClassifierVisibilityKind* on page 327
- *Parameter* on page 328
- *Protocol* on page 329
- *RsClassKind* on page 332
- *RsConcurrency* on page 334
- *Signal* on page 335
- *Feature Classes* on page 336
- *Attribute* on page 338
- *AttributeContainment* on page 340
- *AttributeVisibilityKind* on page 340
- *Operation* on page 340
- *OperationConcurrency* on page 345
- *OperationVisibilityKind* on page 345
- *OwnerScope* on page 346
- *RsOwnerScope* on page 346
- *Collaboration Classes* on page 347
- *AssociationEndRole* on page 350
- *AssociationRole* on page 351
- *CapsuleRole* on page 352
- *CapsuleStructure* on page 353
- *ClassifierRole* on page 356
- *Collaboration* on page 358
- *Connector* on page 364

- *Genericity* on page 367
- *Port* on page 367
- *PortRole* on page 369
- *PortVisibilityKind* on page 370
- *RegistrationMode* on page 370
- *RsGenericity* on page 370
- *RsRegistrationMode* on page 371
- *Common Logical View Enumerations* on page 372
- *RsContainment* on page 372
- *RsVisibilityKind* on page 373*Interaction Classes* on page 374
- *Interaction Classes* on page 374
- *Environment* on page 376
- *Interaction* on page 376
- *InteractionInstance* on page 382
- *Message* on page 385
- *MessageEnd* on page 386
- *RsActionKind* on page 387
- *State Machine Classes* on page 387
- *RsSourceRegionType* on page 388
- *SourceRegionType* on page 389
- *StateMachine* on page 389
- *Transition* on page 390
- *Action Classes* on page 393
- *Action* on page 396
- *ActionMode* on page 398
- *CallAction* on page 399
- *Coregion* on page 399
- *CreateAction* on page 401
- *DestroyAction* on page 401
- *LocalState* on page 402
- *ReplyAction* on page 402
- *RequestAction* on page 402
- *ResponseAction* on page 403
- *ReturnAction* on page 404
- *RsActionMode* on page 404
- *RsSendActionPriority* on page 405
- *SendAction* on page 406
- *SendActionPriority* on page 407
- *TerminateAction* on page 407
- *UninterpretedAction* on page 407
- *Event Classes* on page 407

- *Event* on page 409
- *EventGuard* on page 409
- *PortEvent* on page 411
- *ProtocolRoleEvent* on page 415
- *State Classes* on page 416
- *ChoicePoint* on page 418
- *CompositeState* on page 419
- *FinalState* on page 424
- *InitialPoint* on page 425
- *JunctionContinuationMode* on page 425
- *JunctionPoint* on page 425
- *RsJunctionContinuationMode* on page 427
- *RsStateKind* on page 427
- *StateKind* on page 428
- *StateVertex* on page 429
- *Relation Classes* on page 431
- *ClassDependency* on page 433
- *ClassRelation* on page 434
- *ComponentDependency* on page 435
- *Generalization* on page 436
- *GeneralizationVisibilityKind* on page 438
- *InstantiateRelation* on page 438
- *LogicalPackageDependency* on page 439
- *RealizeRelation* on page 440
- *Relation* on page 442
- *UsesRelationVisibilityKind* on page 444
- *Use Case View Classes* on page 444
- *UseCase* on page 445
- *View Classes* on page 450
- *AnchorNoteView* on page 453
- *Diagram* on page 454
- *NoteView* on page 464
- *RsNoteViewType* on page 466
- *RsStereotypeDisplay* on page 466
- *StereotypeDisplay* on page 467
- *ViewElement* on page 467
- *Class Diagram Classes* on page 475
- *CapsuleView* on page 477
- *ClassDiagram* on page 477
- *ClassView* on page 490
- *ClassifierView* on page 490

- *ProtocolView* on page 492
- *Collaboration Diagram Classes* on page 493
- *CapsuleRoleView* on page 494
- *CollaborationDiagram* on page 496
- *PortRoleView* on page 499
- *PortView* on page 500
- *StructurePerimeterView* on page 500
- *Component Diagram Classes* on page 501
- *ComponentDiagram* on page 502
- *ComponentPackageView* on page 508
- *ComponentView* on page 509
- *Deployment Diagram Classes* on page 509
- *DeploymentDiagram* on page 510
- *Sequence Diagram Classes* on page 513
- *ClassifierRoleView* on page 514
- *CreateMessageView* on page 514
- *InteractionInstanceView* on page 515
- *LifeLineView* on page 515
- *MessageView* on page 516
- *SequenceDiagram* on page 516
- *State Diagram Classes* on page 517
- *BranchPointView* on page 519
- *ChoicePointView* on page 519
- *CompositeStateView* on page 521
- *CoregionView* on page 522
- *FinalStateView* on page 522
- *InitialPointView* on page 523
- *JunctionAdornmentView* on page 523
- *JunctionPointView* on page 524
- *LocalStateOrActionView* on page 525
- *StateDiagram* on page 525
- *StatePerimeterView* on page 527
- *View Property Classes* on page 528
- *LineVertex* on page 529
- *View_FillColor* on page 530
- *View_Font* on page 531
- *View_LineColor* on page 532

Logical Package Structure

The logical package structure is as follows:

Logical View

- *Application Classes* on page 81
- *Extensibility Classes* on page 130
  - *RichTypes* on page 141
- *Model Classes* on page 145
  - *Component View Classes* on page 145
  - *Core Model Classes* on page 178
  - *Deployment View Classes* on page 246
  - *Logical View Classes* on page 267
  - *Association Classes* on page 288
  - *Classifier Classes* on page 299
    - *Feature Classes* on page 336
  - *Collaboration Classes* on page 347
  - *Common Logical View Enumerations* on page 372
  - *Interaction Classes* on page 374
  - *State Machine Classes* on page 387
    - *Action Classes* on page 393
    - *Event Classes* on page 407
    - *State Classes* on page 416
  - *Relation Classes* on page 431
  - *Use Case View Classes* on page 444
- *View Classes* on page 450
  - *Class Diagram Classes* on page 475
  - *Collaboration Diagram Classes* on page 493
  - *Component Diagram Classes* on page 501
  - *Deployment Diagram Classes* on page 509

Sequence Diagram Classes on page 513

State Diagram Classes on page 517

View Property Classes on page 528

Application Classes

Application classes include the following:

AddIn **on page 86**

- Public Attributes

CompanyName : String on page 86

Copyright : String on page 86

EventHandler : Object on page 86

FundamentalTypes : StringCollection on page 86

HelpFilePath : String on page 87

InstallDirectory : String on page 87

MenuFilePath : String on page 87

Name : String on page 87

PropertyFilePath : String on page 87

RootRegistryPath : String on page 87

ServerName : String on page 87

ToolNames : StringCollection on page 88

Version : String on page 88

Activate () : on page 88

AddContextMenuItemForClass (itemType : String, fullCaption : String, internalName : String) : ContextMenuItem on page 88

Deactivate () : on page 89

ExecuteScript (FileName : String) : on page 89

GetContextMenuItemsForClass (itemType : String) : ContextMenuItemCollection on page 90

IsActive () : Boolean on page 90

IsLanguageAddIn () : Boolean on page 90

IsRTAddIn () : Boolean on page 91

ReadSetting (Section : String, Entry : String, Default : String) : String on page 91

WriteSetting (Section : String, Entry : String, Value : String) : Boolean on page 92

AddInManager on page 93

- Public Attributes

AddIns : AddInCollection on page 93

Application on page 93

- Public Attributes

AddInManager : AddInManager on page 94

ApplicationPath : String on page 94

BrowserVisible : Boolean on page 94

CommandLine : String on page 94

CurrentModel : Model on page 94

CurrentWorkspace : Workspace on page 94

Height : Integer on page 95

Left : Integer on page 95

PathMap : PathMap on page 95

ProductName : String on page 95

Top : Integer on page 95

Version : String on page 95

Visible : Boolean on page 95

Width : Integer on page 96

- Public Operations

Add (pElements : ControllableElementCollection, addDirsToo : Boolean, comment : String) : Boolean on page 96

AddDir (pElements : ControllableElementCollection, comment : String) : Boolean on page 97

Browse (pElement : Element, pContext : ModelElement, nLineNumber : Integer) on page 98

CheckIn (pElements : ControllableElementCollection, comment : String) : Boolean on page 99

CheckInDir (pElements : ControllableElementCollection, comment : String) : Boolean on page 100

CheckOut (pElements : ControllableElementCollection) : Boolean on page 101

CompileScriptFile (FileName : String, BinaryName : String, bDebug : Boolean) : on page 101

CreateCollection () : Collection on page 102

ExecuteScript (pFileName : String) : on page 103

Exit () : on page 103

FreeScript (Parameter1 : String) : on page 103

Get (pElements : ControllableElementCollection) : Boolean on page 104

GetLicensedApplication (theKey : String) : Application on page 105

GetObject () : Object on page 105

GetProfileString (Section : String, Entry : String, Default : String) : String on page 106

IsSourceControlEnabled () : Boolean on page 107

LoadScript (Parameter1 : String) : on page 107

NewModel () : Model on page 108

NewScript () : on page 108

OpenExternalDocument (FileName : String) : Boolean on page 109

OpenModel (theModel : String) : Model on page 109*OpenModelAsTemplate (szFileName : String) : Model* on page 110

OpenModelAsTemplate (szFileName : String) : Model on page 110

OpenScript (FileName : String) : on page 110

OpenURL (theURL : String) : Boolean on page 111

OpenWorkspace (FileName : String) : Workspace on page 111

RefreshStatus (pElements : ControllableElementCollection) : Boolean

ReportCodeSync (ocModelElements : Collection, ocContextElements : Collection, ocReplaceStrings : StringCollection) on page 113

Save (bSaveUnits : Boolean) : on page 113

SaveAs (theFile : String, bSaveUnits : Boolean) : on page 114

SaveGenerationResultsAs (filename : String) : Boolean on page 115

SaveLogAs on page 115

SaveWorkspace () : on page 116

SaveWorkspaceAs (FileName : String) : on page 116

SelectObjectsInBrowsers (theObjects : Collection) : on page 117

SetBuildSettings (ShowWarnings : Boolean, VerifyConnectorCardinality : Boolean, VerifyBranchTransitions : Boolean, VerifyDeadUnreachableStates : Boolean, VerifyUntriggeredTransitions : Boolean) : on page 117

UnCheckOut (pElements : ControllableElementCollection) : Boolean on page 118

WriteBuildError (strError : String, pElement : Element, nLineNumber : Integer, bIsWarning : Boolean) : on page 119

WriteBuildOutput (strMessage : String) : on page 120

WriteErrorLog (theMsg : String) : on page 120

WriteErrorLogEx (pszMessage : String, pModelElement : ModelElement, bIsWarning : Boolean) : on page 121

WriteProfileString (Section : String, Entry : String, Value : String) : Boolean on page 121

ContextMenuItem on page 122

- Public Attributes

  Caption : String on page 123

  InternalName : String on page 123

  MenuID : Integer on page 123

  MenuState : MenuState on page 123

MenuState

- *PathMap* on page 124

- Public Operations

  AddEntry (Symbol : String, Path : String, Comment : String) : Boolean on page 124

  DeleteEntry (Symbol : String) : Boolean on page 125

  Get Actual Path (VirtualPath : String) : String

  GetObject () : Object on page 126

  GetVirtualPath (ActualPath : String) : String on page 126

  HasEntry (Symbol : String) : Boolean on page 127

- *RsMenuState* on page 127
  - Public Attributes

    rsDisabled : Integer = 0 on page 128

    rsDisabledAndChecked : Integer = 2 on page 128

    rsDisabledAndUnchecked : Integer = 3 on page 128

    rsDisabledRadioChecked : Integer = 100 on page 128

    rsDisabledRadioUnchecked : Integer = 102 on page 128

    rsEnabled : Integer = 1 on page 128

    rsEnabledAndChecked : Integer = 4 on page 128

    rsEnabledAndUnchecked : Integer = 5 on page 128

    rsEnabledRadioChecked : Integer = 101 on page 128

    rsEnabledRadioUnchecked : Integer = 103 on page 128

Workspace on page 128

- Public Operations

  GetAddInProfileString (theAddIn : AddIn, Entry : String, Default : String) : String on page 128

  WriteAddInProfileString (theAddIn : AddIn, Entry : String, Value : String) : Boolean

AddIn

Description

AddIn class attributes and operations describe and control the characteristics of the AddIns that are part of the currently active Rational Rose RealTime application.

For example, you can

- Find out whether an AddIn is active
- Activate or deactivate an AddIn
- Define the path to the AddIn's menu, property, and help files
- Execute scripts that are specific to the AddIn

Check the lists of attributes and operations for complete information.

Derived from RRTEIObject

Public Attributes

CompanyName : String

Description

Specifies the name of the Company that created the AddIn.

Copyright : String

Description

Specifies copyright information for the AddIn.

EventHandler : Object

Description

Specifies an instance of a custom OLE object implemented by the AddIn developer to provide access to the AddIn from other applications.

FundamentalTypes : StringCollection

Description

Specifies the collection of Fundamental Types that are specific to this AddIn.

HelpFilePath : String

Description

Specifies the path to the AddIn's help file.

InstallDirectory : String

Description

Directory in which the AddIn's executable is installed.

MenuFilePath : String

Description

Specifies the path to the AddIn's menu file.

Name : String

Description

Name of the AddIn.

PropertyFilePath : String

Description

Specifies the path to the AddIn's property file.

RootRegistryPath : String

Description

Specifies the complete registry tree path (from the root) that allows access to the registry entries for this AddIn.

ServerName : String

Description

Specifies the OLE class name that corresponds to the AddIn's EventHandler object.

ToolNames : StringCollection

Description

Specifies the collection of tool names belonging to the AddIn. (Each tool defines its own property sets and corresponds to a tab in the property specification dialog.)

Version : String

Description

Specifies the version number of the AddIn.

Public Operations

Activate () :

Description

Activates the specified AddIn.

Syntax

```
theAddIn.Activate


theAddIn As RoseRT.AddIn
```

AddIn to activate.

AddContextMenuItemForClass (itemType : String, fullCaption : String, internalName : String) : ContextMenuItem

Description

Creates and adds the specified ContextMenuItem to the RoseRT shortcut menu.

Syntax

```
Set theCntxMenuItem = theAddin.AddContextMenuItem(className,
fullCaption, internalName)
```

Parameters

- className - string indicating the type of model element that is in context when the menu option is added to the shortcut menu
- fullCaption - string indicating the caption to display when for the menu option

- internalName - string indicating the name that the item is referenced by in automation

Returns:

A new ContextMenuItem reference to the created item

Deactivate () :

Description

Deactivates the specified AddIn.

Syntax

```
theAddIn.Deactivate
```

```
theAddIn As RoseRT.AddIn
```

AddIn to deactivate.

ExecuteScript (FileName : String) :

Description

Executes the source or compiled image of a script that resides in the AddIn's install directory. This subroutine executes the source or compiled image of a script contained the specified file. You can specify the file without its extension. If the script is currently open in the script editor, Rational Rose RealTime will execute the open script. Otherwise, Rational Rose RealTime will search for the source script (.ebs) and execute it, if found. If not found, Rational Rose RealTime will search for and execute the compiled script (.ebx file).

Syntax

```
theAddIn.ExecuteScript FileName
```

```
theAddIn As RoseRT.AddIn
```

AddIn in which the script is being executed.

```
FileName As String
```

File that contains the script to be executed.

GetContextMenuItemsForClass (itemType : String) : ContextMenuItemCollection

Description

Returns a collection of context menu items based on the requested class.

Syntax

```
Set theItemCollection =  =
theAddin.GetContextMenuItemsForClass(itemType)
```

Parameters

- itemType — string indicating the model element that we want to extract the context menu items for

IsActive () : Boolean

Description

Determines whether the specified AddIn is currently active.

Syntax

```
IsActive = theAddIn.IsActive ()
```

```
IsActive As Boolean
```

Returns a value of True if the specified AddIn is currently active.

```
theAddIn As RoseRT.AddIn
```

AddIn being checked.

IsLanguageAddIn () : Boolean

Description

Determines whether the specified AddIn is a programming language.

Syntax

```
IsLanguage = theAddIn.IsLanguageAddIn ()
```

```
IsLanguage As Boolean
```

Returns a value of True if the specified AddIn is a programming language.

```
theAddIn As RoseRT.AddIn
```

AddIn being checked.

IsRTAddIn () : Boolean

Description

Function that determines whether an AddIn is a Rational Rose RealTime specific AddIn.

Syntax

```
IsRTAddIn = theAddIn.IsRTAddIn()
```

```
IsRTAddIn As Boolean
```

Returns a value of True if the specified AddIn is a Rational Rose RealTime specific AddIn.

```
theAddin As RoseRT.AddIn
```

The instance of AddIn tested as a Rational Rose RealTime AddIn.

ReadSetting (Section : String, Entry : String, Default : String) : String

Description

Retrieves a registry setting for this AddIn.

Syntax

```
theString = theAddIn.ReadSetting (Section, Entry, Default)
```

```
theString As String
```

Returns the actual value of registry setting given its section, entry, and default value. If no corresponding entry exists, returns the specified default value.

```
theAddIn As RoseRT.AddIn
```

The AddIn whose registry entry is being retrieved.

`theSection As String`

Section name of the registry entry. For example: PathMap.

`theEntry As String`

Name of the entry. For example: $SCRIPT_PATH.

`theDefault As String`

Default value of the entry.

WriteSetting (Section : String, Entry : String, Value : String) : Boolean

Description

Creates a registry entry for this AddIn.

Syntax

```
IsWritten = theAddIn.WriteSetting (Section, Entry, Value)
```

`IsWritten As Boolean`

Returns a value of True when the entry is successfully added to the registry.

`theAddIn As RoseRT.AddIn`

AddIn for which the registry setting is being created.

`theSection As String`

User-defined section name for the custom entry.

`theEntry As String`

User-defined entry name.

`theValue As String`

User-defined default value for the custom entry.

AddInManager

Description

The AddInManager class has a single attribute, the AddIns attribute, which contains the collection of AddIns available to the currently active Rational Rose RealTime executable. The AddInManager class inherits all RRTEIObject operations, but has no operations of its own.

Derived from RRTEIObject

Public Attributes

AddIns : AddInCollection

Description

Specifies the collection of AddIns managed by the RoseAddInManager

Application

Description

Use the application class to

- Create a new model
- Select an existing model as the current model
- Determine the characteristics of the Rational Rose RealTime application being controlled by your script

Here are a few of the application characteristics you can control with application class attributes and operations:

- How (and if) the Rational Rose RealTime application appears on the computer screen while the script is running
- The size and position of the Rational Rose RealTime application window
- Whether to write errors to the error log

Derived from RoseBase

Public Attributes

AddInManager : AddInManager

Description

Specifies the Rose AddIn Manager belonging to the currently active Rational Rose RealTime executable.

ApplicationPath : String

Description

Specifies the path to the Rational Rose RealTime application to execute.

BrowserVisible : Boolean

Description

Controls whether the Rational Rose RealTime application is visible on the computer screen.

CommandLine : String

Description

Returns the command line option string that is passed when the Rose executable is run.

CurrentModel : Model

Description

Specifies the model that is currently open in Rational Rose RealTime.

CurrentWorkspace : Workspace

Description

Specifies the workspace that is currently open in Rational Rose RealTime.

Height : Integer

Description

Specifies the height of the main window.

Left : Integer

Description

Specifies the distance between the left side of the main window and the left side of the screen.

PathMap : PathMap

Description

Returns the path map defined for the current Rose application.

ProductName : String

Description

Returns the product name for the currently active Rose RealTime application.

Top : Integer

Description

Specifies the distance between the top of the main window and top of the screen.

Version : String

Description

Returns the version of the currently active Rose RealTime application. Corresponds to the information provided when you select About from the Help menu in Rose RealTime.

Visible : Boolean

Description

Controls whether the Rose RealTime application is visible on the computer screen.

Width : Integer

Description

Specifies the width of the main window.

Public Operations

Add (pElements : ControllableElementCollection, addDirsToo : Boolean, comment : String) : Boolean

Description

Adds a collection of ControllableElement to Source Control.

Syntax

```
Added = theApplication.Add( pElements, AddDirsToo, comment )
```

`Added As Boolean`

Returns a value of True if Controllable Elements in pElements Collection were added successfully to Source Control.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements to add to Source Control.

`AddDirsToo As Boolean`

Always False. Reserved for future use.

`comment As String`

Comments to provide to Source Control server for the operation.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.Add( theCECollection, True, "My Add Comment" )

Set theCECollection = Nothing
```

AddDir (pElements : ControllableElementCollection, comment : String) : Boolean

Description

Adds the directories associated with a collection of Controllable Elements to source control. This only applies to Packages. The only circumstance under which this is needed is when a model is placed under source control without all elements controlled. In this situation, the model's directory is not source controlled. If the model is subsequently controlled, then the model's directory must be added to source control before any of the model's child elements can be added to source control.

Syntax

```
Added = theApplication.AddDir( pElements, comment )
```

`Added As Boolean`

Returns a value of True if the directories associated with the Controllable Elements in pElements Collection were added successfully to Source Control.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements to add to

Source Control.

`comment As String`

Comments to provide to Source Control server for the operation.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.AddDir( theCECollection, "My AddDir Comment" )

Set theCECollection = Nothing
```

Browse (pElement : Element, pContext : ModelElement, nLineNumber : Integer)

Description

Opens the diagram & spec sheet corresponding to the given model element & context.

Syntax

```
theApplication.Browse( pElement, pContext, nLineNumber )
```

```
theApplication As RoseRT.Application
```

The running instance of Application.

```
pElement As RoseRT.Element
```

The element to browse to.

```
pContext As RoseRT.ModelElement
```

The context of the given element.

```
nLineNumber As Integer
```

The line of code to highlight. (if appropriate).

CheckIn (pElements : ControllableElementCollection, comment : String) : Boolean

Description

CheckIn a collection of ControllableElement in Source Control.

Syntax

```
CheckedIn = theApplication.CheckIn( pElements, comment )
```

`CheckedIn As Boolean`

Returns a value of True if Controllable Elements in pElements Collection were checked in successfully to Source Control.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements to checkin in Source Control.

`comment As String`

Comments to provide to Source Control server for the operation.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.CheckIn( theCECollection, "My CheckIn Comment" )

Set theCECollection = Nothing
```

CheckInDir (pElements : ControllableElementCollection, comment : String) : Boolean

Description

CheckIn directories used for child controllable element storage of a collection of ControllableElement's in Source Control.

Syntax

```
CheckedChildDirIn = theApplication.CheckInDir( pElements, comment )
```

`CheckedIn As Boolean`

Returns a value of True if the child directory of Controllable Elements in pElements Collection were checked in successfully to Source Control.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements whose child directory are to be checked in Source Control.

`comment As String`

Comments to provide to Source Control server for the operation.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.CheckInDir( theCECollection, "My CheckIn Comment" )

Set theCECollection = Nothing
```

CheckOut (pElements : ControllableElementCollection) : Boolean

Description

CheckOut a collection of ControllableElement from Source Control.

Syntax

```
CheckedOut = theApplication.CheckOut( pElements )
```

`CheckedOut As Boolean`

Returns a value of True if Controllable Elements in pElements Collection were checked out successfully from Source Control.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements to checkout from Source Control.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.CheckOut( theCECollection )

Set theCECollection = Nothing
```

CompileScriptFile (FileName : String, BinaryName : String, bDebug : Boolean) :

Description

Compiles the script contained in the specified file.

Syntax

```
theApplication.CompileScriptFile theFileName, theBinaryName, Debug
```

`theApplication As RoseRT.Application`

Instance of the Rose application in which the script is being compiled.

`theFileName As String`

Name of the file that contains the script being compiled; include the .ebs file extension.

`theBinaryName As String`

Name of the binary file in which to save the compiled script; use the .ebx file extension.

`Debug As Boolean`

Set to True to embed the script's source code in the compiled file. This allows the script debugger to display the source code when it enters external modules.

CreateCollection () : Collection

Description

Returns a new empty generic collection.

Syntax

```
Set theCollection = theApplication.CreateCollection()
```

`theCollection As RoseRT.Collection`

Newly created generic empty collection.

`theApplication As RoseRT.Application`

Instance of the Rose RealTime application owning the returned collection.

ExecuteScript (pFileName : String) :

Description

Executes the source or compiled image of a script contained the specified file. You can specify the file without its extension. If the script is currently open in the script editor, Rose RealTime will execute the open script. Otherwise, Rose RealTime will search for the source script (.ebs) and execute it, if found. If not found, Rose RealTime will search for and execute the compiled script (.ebx file).

Syntax

```
theApplication.ExecuteScript theFileName
```

```
theApplication As RoseRT.Application
```

Instance of the Rose application in which the script is being executed.

```
theFileName As String
```

Name of the file that contains the script to execute.

Exit () :

Description

Exits the Rose RealTime application.

Syntax

```
theApplication.Exit
```

```
theApplication As RoseRT.Application
```

Instance of the Rose application being exited.

FreeScript (Parameter1 : String) :

Description

Unloads the source or compiled image of a script contained in the specified file. Specify the file without its extension and Rose RealTime frees the source script (.ebs), if found. If not found, Rose RealTime frees the compiled script (.ebx file).

Notes

- This subroutine is only valid for Rose Script; it does not exist in Rose RealTime Automation
- Every LoadScript call should have a subsequent FreeScript call. See LoadScript Method for more information.

Syntax

```
theApplication.FreeScript theFileName
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime from which the script is being unloaded.

```
theFileName As String
```

The name of the file that contains script to unload. Do not specify a file extension.

Get (pElements : ControllableElementCollection) : Boolean

Description

Get a collection of ControllableElement from Source Control.

Syntax

GetDone = theApplication.Get(pElements)

```
GetDone As Boolean
```

Returns a value of True if Controllable Elements in pElements Collection were Get successfully to Source Control.

```
theApplication As RoseRT.Application
```

The running instance of Application.

```
pElements As RoseRT.ControllableElementCollection
```

The collection containing the ControllableElements to get from Source Control.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.Get( theCECollection )

Set theCECollection = Nothing
```

GetLicensedApplication (theKey : String) : Application

Description

Retrieves an instance of the licensed application given the application's licensing key.

Syntax

```
Set theInstance = theApplication.GetLicensedApplication (theKey)
```

`theInstance As RoseRT.Application`

Returns the instance of the licensed application.

`theApplication As RoseRT.Application`

Currently active application.

`theKey As String`

Licensing key for the application being retrieved.

GetObject () : Object

Description

Retrieves the OLE automation interface object associated with the specified application.

Note: This operation is only valid for Rose RealTime Script; it does not exist in Rose RealTime Automation.

Syntax

```
Set theOLEObject = theApplication.GetObject ( )
```

`theOLEObject As RoseRT.Object`

Returns the OLE automation interface object associated with the application.

`theApplication As RoseRT.Application`

Instance of the Rose application whose OLE automation interface object is being returned.

GetProfileString (Section : String, Entry : String, Default : String) : String

Description

Retrieves a profile string entry in the RoseRT.ini file, given a section, entry, and default value.

Syntax

```
Set theProfileString = theApplication.GetProfileString (theSection,
the Entry, theDefault)
```

`theProfileString As String`

Returns the profile string that corresponds to the given section, entry, and default value.

`theApplication As RoseRT.Application`

Currently active application and therefore the application whose RoseRT.ini file entry is being retrieved.

`theSection As String`

Name of the RoseRT.ini file section from which the profile string is being retrieved. For example: [PathMap]

`theEntry As String`

The name of the RoseRT.ini file entry whose profile string is being retrieved. For example: $SCRIPT_PATH

```
theDefault As String
```

Default value of the entry being retrieved. In the [PathMap] $SCRIPT_PATH example, the default value is the path to the folder that contains the scripts being called by the application.

IsSourceControlEnabled () : Boolean

Description

Determines whether Source Control is enabled for the current Workspace.

Syntax

```
SourceControlEnabled = theApplication.IsSourceControlEnabled()
```

```
SourceControlEnabled As Boolean
```

Returns a value of True if Source Control is enabled for the current Workspace.

```
theApplication As RoseRT.Application
```

The running instance of Application.

LoadScript (Parameter1 : String) :

Description

Loads the source or compiled image of a script contained in the specified file. You can specify the file without its extension and Rose RealTime will load the source script (.ebs), if found. If not found, Rose RealTime will load the compiled script (.ebx file).

Notes

- This subroutine is only valid for Rose RealTime Script; it does not exist in Rose RealTime Automation.
- When finished with the script, you should make a call to FreeScript. Because scripts contain reference counting information, if you call LoadScript on a given script 10 times, you should subsequently call FreeScript 10 times; otherwise, the script will not be unloaded.

Syntax

```
theApplication.LoadScript theFileName
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application in which the script is being loaded.

```
theFileName As String
```

Name of the file that contains the script. Do not specify a file extension.

NewModel () : Model

Description

Creates a new Rose RealTime model and returns it as a model object.

Syntax

```
Set theModel = theApplication.NewModel ()
```

```
theModel As RoseRT.Model
```

Contains the newly created Rose RealTime model.

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application in which the model is being created.

NewScript () :

Description

Opens a script editor window in which to create a new script.

Note: This subroutine is only valid for Rose RealTimeScript; it does not exist in Rose RealTime Automation.

Syntax

```
theApplication.NewScript
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application in which the new script is being created.

OpenExternalDocument (FileName : String) : Boolean

Description

Opens an external document, given a fully qualified name of the file that contains the document.

Syntax

```
IsOpen = theApplication.Open (theFileName)
```

`IsOpen As Boolean`

Returns a value of true when the specified document is successfully opened.

`theApplication As RoseRT.Application`

Currently active application.

`theFileName As String`

Fully qualified file name or the URL that contains the external document.

OpenModel (theModel : String) : Model

Description

Opens a Rose RealTime model and returns it as a model object.

Syntax

```
Set theModel = theApplication.OpenModel (theName)
```

`theModel As RoseRT.Model`

Contains the model being opened.

`theApplication As RoseRT.Application`

Instance of the Rose RealTime application from which the model is being retrieved.

`theName As String`

Name of the model being opened.

OpenModelAsTemplate (szFileName : String) : Model

Description

Retrieves an existing model to be used as a template from which to create a new model.

Syntax

```
Set theModel = theApplication.OpenModelAsTemplate (FileName)
```

```
theModel As RoseRT.Model
```

Returns the model contained in the specified file.

```
theApplication As RoseRT.Application
```

Currently active application.

```
theFileName As String
```

Name of the file that contains the model being returned.

OpenScript (FileName : String) :

Description

Opens the source or compiled image of a script contained in the specified file in the script editor window. You can specify the file without its extension and Rose RealTime will search for the source script (.ebs) and open it, if found. If not found, Rose RealTime will search for and open the compiled script (.ebx file).

Note: This subroutine is only valid for Rose RealTime Script; it does not exist in Rose RealTime Automation.

Syntax

```
theApplication.OpenScript FileName
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application in which the script is being opened.

`FileName As String`

Name of the script file being opened.

OpenURL (theURL : String) : Boolean

Description

Opens a URL, given the URL string.

Syntax

```
IsOpen = theApplication.Open (theURL)
```

`IsOpen As Boolean`

Returns a value of true when the specified URL is successfully opened.

`theApplication As RoseRT.Application`

Currently active application.

`theURL As String`

URL that contains the external document.

OpenWorkspace (FileName : String) : Workspace

Description

Opens a Rose RealTime workspace and the model associated with it.

Syntax

```
Set theWorkspace = theApplication.OpenWorkspace (FileName)
```

`theWorkspace As RoseRT.Workspace`

Contains the workspace being opened.

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application from which the workspace is being retrieved.

```
FileName As String
```

Name of the workspace being opened.

RefreshStatus (pElements : ControllableElementCollection) : Boolean

Description

Refresh the Source Control status of a collection of ControllableElement.

Syntax

```
Refreshed = theApplication.RefreshStatus( pElements )
```

```
Refreshed As Boolean
```

Returns a value of True if the Source Control status of the Controllable Elements in pElements Collection were Refreshed successfully.

```
theApplication As RoseRT.Application
```

The running instance of Application.

```
pElements As RoseRT.ControllableElementCollection
```

The collection containing the ControllableElements whose Source Control status are to be refreshed.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.RefreshStatus( theCECollection )
```

```
Set theCECollection = Nothing
```

ReportCodeSync (ocModelElements : Collection, ocContextElements : Collection, ocReplaceStrings : StringCollection)

Description:

Updates the model elements with the new code corresponding to changes in the generated code.

Syntax:

```
theApplication.ReportCodeSync( ocModelElements, ocContextElements,
ocReplaceStrings )
```

`ocModelElements As Collection`

Contains the model elements that need to be code synchronized with the modified generated code.

`ocContextElements As Collection`

Contains the elements that are the contexts for the elements in the ocModelElementsCollection. This collection corresponds one to one with the ModelElements collection.

`ocReplaceStrings As StringCollection`

Contains the new code changes that need to be code synchronized back to the original model elements. This collection corresponds one to one with the model element collection.

Save (bSaveUnits : Boolean) :

Description

Saves the current Rose RealTime model.

Note: This operation is not valid if any of the following is true:

- The file containing the Rose RealTime model is ReadOnly
- The file containing the Rose RealTime model is unnamed
- SaveUnits is True and any Unit cannot be saved

Syntax

```
theApplication.Save SaveUnits
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application whose current model is being saved.

```
SaveUnits As Boolean
```

Indicates whether the current model is comprised of controlled units.

SaveAs (theFile : String, bSaveUnits : Boolean) :

Description

Names and saves the current Rose RealTime model.

Note: This operation is not valid under the following conditions:

- The file containing the Rose RealTime model is ReadOnly
- The file containing the Rose RealTime model is unnamed
- SaveUnits is True and any Unit cannot be saved

Syntax

```
theApplication.SaveAs theName, SaveUnits
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application whose current model is being saved.

```
theName As String
```

Name of the model being saved.

```
SaveUnits As Boolean
```

Indicates whether the current model is comprised of controlled units.

SaveGenerationResultsAs (filename : String) : Boolean

Description

Saves the Code Generation Results in a file

Syntax

```
Saved = theApplication.RefreshStatus( filename )
```

`Saved As Boolean`

Returns a value of True if the Code Generation Results were saved successfully.

`theApplication As RoseRT.Application`

The running instance of Application.

`filename As String`

The filename of the file to save Code Generation Results to.

SaveLogAs

Description

Saves the error log in a file

Syntax

```
Saved = theApplication.SaveLogAs( filename )
```

`Saved As Boolean`

Returns a value of True if the error log was saved successfully

`theApplication As RoseRT.Application`

The running instance of Application

`filename As String`

The filename of the file to save thet error log to

SaveWorkspace () :

Description

Saves the current workspace.

Note: This operation is not valid if any of the following is true:

- The file containing the workspace is ReadOnly
- The Rose RealTime model is unnamed

Syntax

```
theApplication.SaveWorkspace
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application whose current workspace is being saved.

SaveWorkspaceAs (FileName : String) :

Description

Names and saves the current workspace.

Note: This operation is not valid under the following conditions:

- The file with the passed in filename already exist
- The Rose RealTime model is unnamed

Syntax

```
theApplication.SaveWorkspaceAs FileName
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application whose current workspace is being saved.

```
FileName As String
```

Name of the workspace being saved.

SelectObjectsInBrowsers (theObjects : Collection) :

Description

Selects objects in visible browsers.

Syntax

```
theApplication.SelectObjectsInBrowsers( theObjectCollection )
```

`theApplication As RoseRT.Application`

The running instance of Application.

`theObjectCollection As RoseRT.Collection`

The collection of objects to select in visible browsers.

Example

```
Dim theObjects As RoseRT.Collection
Set theObjects = theApplication.CreateCollection()

b = theObjectCollection.Add( RoseRTApp.CurrentModel )
b = theObjectCollection.Add( RoseRTApp.CurrentModel.RootLogicalPackage
)

b = RoseRTApp.SelectObjectsInBrowsers( theObjects )

Set theObjects = Nothing
```

SetBuildSettings (ShowWarnings : Boolean, VerifyConnectorCardinality : Boolean, VerifyBranchTransitions : Boolean, VerifyDeadUnreachableStates : Boolean, VerifyUntriggeredTransitions : Boolean) :

Description

Allows configuration of common build settings that will be used when building any component.

Syntax

```
theApplication.SetBuildsSettings( ShowWarnings,
VerifyConnectorCardinality, VerifyBranchTransitions,
VerifyDeadUnreachableStates, VerifyUntriggeredTransitions )
```

```
theApplication As RoseRT.Application
```

The running instance of Application.

```
ShowWarnings As Boolean
```

Whether to show warning.

```
VerifyConnectorCardinality As Boolean
```

Whether to test if cardinalities on both side of a connection are equivalents.

```
VerifyBranchTransitions As Boolean
```

Whether to check for missing true or false transitions on choice points.

```
VerifyDeadUnreachableStates As Boolean
```

Whether to check for all states that are not reachable in a state diagram and for all states that cannot be exited.

```
VerifyUntriggeredTransitions As Boolean
```

Whether to check for transitions with no triggering event

UnCheckOut (pElements : ControllableElementCollection) : Boolean

Description

Undo a CheckOut operation for a collection of ControllableElement.

Syntax

```
UndidCheckedOut = theApplication.UndoCheckOut( pElements )
```

```
UndidCheckedOut As Boolean
```

Returns a value of True if Controllable Elements in pElements Collection had their CheckOut operation successfully undone.

`theApplication As RoseRT.Application`

The running instance of Application.

`pElements As RoseRT.ControllableElementCollection`

The collection containing the ControllableElements to undo the checkout operation from.

Example

```
Dim theCECollection As RoseRT.Collection
Set theCECollection = theApplication.CreateCollection()

b = theCECollection.Add( RoseRTApp.CurrentModel )
b = RoseRTApp.UndoCheckOut( theCECollection )

Set theCECollection = Nothing
```

WriteBuildError (strError : String, pElement : Element, nLineNumber : Integer, bIsWarning : Boolean) :

Description

Writes an entry in the error/warning list section of the build log window.

Syntax

```
theApplication.WriteBuildError( strError, pElement, nLineNumber,
bIsWarning )
```

`theApplication As RoseRT.Application`

The running instance of Application.

`strError As String`

Description of error/warning.

```
pElement As RoseRT.Element
```

The element that owns the source code where an error/warning was detected.

```
nLineNumber As Integer
```

The line number where the error/warning was detected in source code

```
bIsWarning As Boolean
```

Whether the new entry represents a warning or an error

WriteBuildOutput (strMessage : String) :

Description

Writes a message to the output section of the build log window.

Syntax

```
theApplication.WriteBuildOutput( strMessage )
```

```
theApplication As RoseRT.Application
```

The running instance of Application.

```
strMessage As String
```

Message to output.

WriteErrorLog (theMsg : String) :

Description

Writes an error message to a log window.

Syntax

```
theApplication.WriteErrorLog theMessage
```

```
theApplication As RoseRT.Application
```

Instance of the Rose RealTime application for which errors are being logged.

`theMessage As String`

Message text to write to the error log window.

WriteErrorLogEx (pszMessage : String, pModelElement : ModelElement, bIsWarning : Boolean) :

Description

Writes an entry in the error log window.

Syntax

```
theApplication.WriteErrorLogEx( pszMessage, pModelElement, bIsWarning
)
```

`theApplication As RoseRT.Application`

The running instance of Application.

`strMessage As String`

Description of error/warning.

`pModelElement As RoseRT.ModelElement`

The model element related to the error/warning.

`bIsWarning As Boolean`

Whether the new entry represents a warning or an error

WriteProfileString (Section : String, Entry : String, Value : String) : Boolean

Description

Retrieves a profile string entry in the RoseRT.ini file, given a section, entry, and default value.

Syntax

```
IsWritten = theApplication.WriteProfileString (Section, Entry, Value)
```

`IsWritten As Boolean`

Returns a value of true when the specified ProfileString is successfully written to the Rose.ini file.

`theApplication As RoseRT.Application`

Currently active application and therefore the application whose RoseRT.ini file entry is being written.

`theSection As String`

Name of the RoseRT.ini file section to which the profile string is being written. For example: [PathMap]

`theEntry As String`

The name of the RoseRT.ini file entry whose profile string is being written. For example: $SCRIPT_PATH

`theValue As String`

Value of the entry being written. In the [PathMap] $SCRIPT_PATH example, the value is the actual path to the folder that contains the scripts being called by the application.

ContextMenuItem

Description

This class represents a context menu option that was added through RRTEI by an addin. References to this class are returned by the AddContextMenuItemForClass method of AddIn

Derived from RRTEIObject

Public Attributes

Caption : String

Description

The text that is displayed when the item is added to a context menu

InternalName : String

Description

The string that is returned to the automation server when an item is selected

MenuID : Integer

Description

The internal ID used to index the menu item for the class it corresponds to

MenuState : MenuState

Description

The state the menu item is displayed in. See the RsMenuState enumeration for possible values.

MenuState

Description

Rich type used to determine the state of a context menu. Valid values are defined in the RsMenuState enumeration.

Derived from RichType

PathMap

Description

Use the PathMap class to create and edit path map entries for the current model. For example, you can create entries to define paths to controlled units, to scripts executed from the Rose RealTime menu, and to the root directory for a multi-user project. Executing PathMap class operations is equivalent to updating the PathMap dialog in the Rose RealTime user interface. There are no attributes associated with the PathMap class.

Derived from RoseBase

Public Operations

AddEntry (Symbol : String, Path : String, Comment : String) : Boolean

Description

Adds an entry to the current application's PathMap definition.

Syntax

```
IsAdded = thePathMap.AddEntry (theSymbol, theActualPath, theComment))
```

```
IsAdded As Boolean
```

Returns a value of true when the entry is successfully added.

```
thePathMap As RoseRT.PathMap
```

PathMap to which the entry is being added.

```
thelSymbol As String
```

Virtual symbol being added to the PathMap. For example, $SCRIPT_PATH

```
theActualPath As String
```

Actual path to which the virtual symbol refers.

```
theComment As String
```

Description of the PathMap entry being added.

DeleteEntry (Symbol : String) : Boolean

Description

Deletes an entry from the current application's PathMap definition.

Syntax

```
IsDeleted = thePathMap.DeleteEntry (theSymbol)
```

```
IsDeleted As Boolean
```

Returns a value of true when the entry is successfully deleted.

```
thePathMap As RoseRT.PathMap
```

PathMap to which the entry is being added.

```
theSymbol As String
```

Virtual symbol for the entry being deleted from the PathMap. For example, $SCRIPT_PATH

Get Actual Path (VirtualPath : String) : String

Description

Retrieves from the PathMap the actual path that corresponds to the given virtual symbol.

Syntax

```
theActualPath = thePathMap.GetActualPath (theSymbol)
```

```
theActualPath As String
```

Returns the actual path given the virtual symbol.

```
thePathMap As RoseRT.PathMap
```

PathMap from which to retrieve the actual path.

```
theSymbol As String
```

Virtual symbol whose corresponding actual path is being retrieved.

GetObject () : Object

Description

Retrieves the object's OLE interface object.

Note: This function is only valid for Rose RealTime Script; it has no meaning in Rose RealTime Automation.

Syntax

```
Set theOLEObject = thePathMap.GetObject ( )
```

```
theOLEObject As RoseRT.Object
```

Returns the OLE automation interface object associated with the specified object.

```
thePathMap As RoseRT.PathMap
```

Instance of the object whose OLE interface object is being returned.

GetVirtualPath (ActualPath : String) : String

Description

Retrieves the virtual path that corresponds to the given actual path.

Syntax

theString = thePathMap.GetVirtualPath (theActualPath)

```
theVirtualPath As String
```

Returns the virtual path given the actual path.

```
thePathMap As RoseRT.PathMap
```

PathMap from which to retrieve the virtual path.

```
theActualPath As String
```

Actual path whose corresponding virtual path is being retrieved.

HasEntry (Symbol : String) : Boolean

Description

Checks the PathMap for an entry based on the given virtual path symbol.

Syntax

```
HasEntry = thePathMap.HasEntry (theSymbol)
```

```
HasEntry As Boolean
```

Returns a value of True if the PathMap has an entry for the given virtual path symbol.

```
thePathMap As RoseRT.PathMap
```

PathMap being checked.

```
theSymbol As String
```

Virtual symbol to search for in the PathMap.

RsMenuState

Description

Enumeration used to set the Value property of the MenuState rich type. Values determine what state add-in context menu items are displayed in.

Public Attributes

rsDisabled : Integer = 0

rsDisabledAndChecked : Integer = 2

rsDisabledAndUnchecked : Integer = 3

rsDisabledRadioChecked : Integer = 100

rsDisabledRadioUnchecked : Integer = 102

rsEnabled : Integer = 1

rsEnabledAndChecked : Integer = 4

rsEnabledAndUnchecked : Integer = 5

rsEnabledRadioChecked : Integer = 101

rsEnabledRadioUnchecked : Integer = 103

Workspace

Description

Represents a workspace file. The workspace maintains information about the current model, open windows and window positions, etc. The workspace information is stored in a separate file (a .rtwks file). This class allows clients to inquire and modify settings saved within the workspace file.

Derived from RoseBase

Public Operations

GetAddInProfileString (theAddIn : AddIn, Entry : String, Default : String) : String

Description

Retrieves a profile string entry for an Add-In in the workspace, given an Add-In, and entry and a default value.

Syntax

```
Set theProfileString = theWorkspace.GetAddInProfileString (theAddIn,
Entry, Default)
```

`theProfileString As String`

Returns the profile string that corresponds to the given Add-In, entry, and default value.

`theWorkspace As RoseRT.Workspace`

Workspace whose entry is being retrieved.

`theAddIn As RoseRT.AddIn`

Add-In whose entry profile string is being retrieved for.

`theEntry As String`

The name of the entry whose profile string is being retrieved.

`theDefault As String`

Default value of the entry being retrieved. This is the string returned if the entry does not exists in the workspace for the Add-In.

WriteAddInProfileString (theAddIn : AddIn, Entry : String, Value : String) : Boolean

Description

Write a profile string entry for an Add-In in the workspace, given an Add-In, an entry, and a value.

Note: This operation is not valid if any of the following is true:

- The file containing the workspace is ReadOnly
- The Rose RealTime model is unnamed

Syntax

```
IsWritten = theWorkspace.WriteAddInProfileString (theAddIn, Entry,
Value)
```

`IsWritten As Boolean`

Returns a value of true when the specified ProfileString is successfully written in the workspace.

`theWorkspace As RoseRT.Application`

Workspace that gets an entry written to.

`theAddIn As RoseRT.AddIn`

Add-In whose entry profile string is being written to.

`theEntry As String`

The name of the entry whose profile string is being written.

`theValue As String`

Value of the entry being written.

Extensibility Classes

Extensibility classes include

- *Collection* on page 131
    - Public Attributes

      Count : Integer on page 131
    - Public Operations

      Add (theObject : RoseBase) : on page 132

      AddCollection (theCollection : Collection) : on page 132

      Exists (pObject : RoseBase) : Boolean on page 133

      FindFirst (Name : String) : Integer on page 133

      FindNext (iCurID : Integer, Name : String) : Integer on page 134

      GetAt (Index : Integer) : RoseBase on page 135

GetFirst (Name : String) : RoseBase on page 135

GetObject () : Object on page 136

GetWithUniqueID (UniqueID : String) : Object on page 137

IndexOf (theObject : RoseBase) : Integer on page 137

Remove (theObject : RoseBase) : on page 138

RemoveAll () : on page 139

- *RoseBase* on page 139
  - Public Attributes

    GetObject () : Object on page 139
- *RRTEIObject* on page 140
  - Public Operations

    IdentifyClass () : String on page 140

Collection

Description

For most elements of a RoseRT model there is a corresponding collection. So, for example, for every class there is a class collection; for every logical package there is a logical package collection; for every property, there is a property collection, and so on.

RoseRT extensibility provides a set of properties and methods that allow you to access a particular element in any given collection.

Derived from RoseBase

Public Attributes

Count : Integer

Description

Number of elements in the collection.

Public Operations

Add (theObject : RoseBase) :

Description

Adds an object to a collection.

Syntax

```
theCollection.Add theObject
```

```
theCollection As RoseRT.Collection
```

Collection to which the object is being added.

```
theObject As Object
```

Object being added to the collection.

AddCollection (theCollection : Collection) :

Description

Adds a collection of objects to a collection.

Note: The objects are added as individual objects, not as a collection. For this reason, should you need to remove one or more of these objects from the destination collection, you can simply use the Remove or RemoveAll method.

Syntax

```
theCollection.AddCollection theObjectCollection
```

```
theCollection As RoseRT.Collection
```

Collection to which the collection of objects is being added.

```
theObjectCollection As Collection
```

Collection whose objects are being added.

Exists (pObject : RoseBase) : Boolean

Description

Checks for the existence of an object in a collection

Syntax

```
Exists = theCollection.Exists (theObject)
```

`Exists As Boolean`

Returns a value of True if the object exists in the collection.

`theCollection As RoseRT.Collection`

The collection being checked.

`theObject As Object`

Instance of the object whose existence is being checked.

FindFirst (Name : String) : Integer

Description

Returns the index (position) of the first instance of the named object from a collection.

Note: To retrieve the object itself, use the GetAt method and specify the index returned by this method.

Syntax

```
Set theIndex = theCollection.FindFirst (theName)
```

`theIndex  As Integer`

Returns the index of the first instance of the named object in the collection. Returns a value of 0 if the named object is not found.

`theObject As RoseRT.Collection`

Collection from which the index is being retrieved.

```
theName As String
```

Name of the object whose index is being retrieved.

See also

FindNext (iCurID : Integer, Name : String) : Integer on page 134

IndexOf (theObject : RoseBase) : Integer on page 137

GetFirst (Name : String) : RoseBase on page 135

FindNext (iCurID : Integer, Name : String) : Integer

Description

When iterating through a collection, this function retrieves the index (position) of the next instance of the named object, given the index of the current instance.

Note: To retrieve the object itself, use the GetAt method and specify the index returned by this method.

Syntax

```
NextIndex = theCollection.FindNext (CurrentIndex, theName)
```

```
NextIndex As Integer
```

Returns the index of the next instance of an object from the collection.

Returns a value of 0 if the named object is not found.

```
theCollection As RoseRT.Collection
```

Collection from which the next index is being retrieved.

```
CurrentIndex As Integer
```

Index of the current object instance in the collection.

```
theName As String
```

Name of the object whose index is being retrieved.

See also

FindFirst (Name : String) : Integer on page 133

GetFirst (Name : String) : RoseBase on page 135

IndexOf (theObject : RoseBase) : Integer on page 137

GetFirst (Name : String) : RoseBase on page 135

GetAt (Index : Integer) : RoseBase

Description

Retrieves a particular object from a collection, given the object's position in the collection.

Syntax

```
Set theObject = theCollection.GetAt (theIndex)
```

Note: To get the index of the object, use the IndexOf, FindFirst or FindNext method.

```
theObject As Object
```

Returns an object from the collection.

```
theCollection As RoseRT.Collection
```

Collection from which to retrieve the object.

```
theIndex As Integer
```

Index (position) of the object in the collection.

See also

FindFirst (Name : String) : Integer on page 133

FindNext (iCurID : Integer, Name : String) : Integer on page 134

IndexOf (theObject : RoseBase) : Integer on page 137

GetFirst (Name : String) : RoseBase on page 135

GetFirst (Name : String) : RoseBase

Description

Retrieves the first instance of the named object from a collection.

Syntax

```
Set theObject = theCollection.GetFirst (theName)
```

`theObject As Object`

Returns the first instance of the named object from the collection.

`theCollection As RoseRT.Collection`

Collection from which to retrieve the object.

`theName As String`

Name of the object to retrieve.

See also

FindFirst (Name : String) : Integer on page 133

FindNext (iCurID : Integer, Name : String) : Integer on page 134

IndexOf (theObject : RoseBase) : Integer on page 137

GetObject () : Object

Description

Retrieves the OLE object associated with a specified collection.

Note: This function is only valid for Rose Script; it does not exist in Rose Automation.

Syntax

```
Set theOLEObject = theCollection.GetObject ( )
```

`theOLEObject As Object`

Returns the OLE automation interface object associated with the specified object.

`theCollection As RoseRT.Collection`

Instance of the object whose interface object is being returned.

GetWithUniqueID (UniqueID : String) : Object

Description

Retrieves an object from a collection, given the object's unique ID. This is simpler than iterating through the collection to find a named or indexed object. Every element in a model has a unique ID. You cannot set this ID, but you can retrieve it.

Syntax

```
Set theObject = theCollection.GetWithUniqueID (theUniqueID)
```

`theObject As Object`

Returns the object whose unique ID you specify.

`theCollection As RoseRT.Collection`

Collection from which to retrieve the object.

`theUniqueID As String`

UniqueID of the object to retrieve.

See also

FindFirst (Name : String) : Integer on page 133

FindNext (iCurID : Integer, Name : String) : Integer on page 134

IndexOf (theObject : RoseBase) : Integer on page 137

IndexOf (theObject : RoseBase) : Integer

Description

Retrieves the index (position) of an instance of an object in a collection.

Syntax

Set theIndex = theCollection.IndexOf (theObject)

theIndex As Integer

Returns the index (position) of the given objectReturns a value of 0 if the class is not found.

theCollection As RoseRT.Collection

Collection from which the index is being retrieved.

theObject As Object

Instance of the object whose index is being retrieved.

See also

FindFirst (Name : String) : Integer on page 133

FindNext (iCurID : Integer, Name : String) : Integer on page 134

GetFirst (Name : String) : RoseBase on page 135

Remove (theObject : RoseBase) :

Description

Removes an object from a collection.

Syntax

```
theCollection.Remove theObject
```

```
theCollection As RoseRT.Collection
```

Collection from which the class is being removed.

```
theObject As Object
```

Object being removed from the collection.

See also

RemoveAll () : on page 139

RemoveAll () :

Description

Removes all objects from a collection.

Syntax

```
theCollection.RemoveAll
```

```
theCollection As RoseRT.Collection
```

Collection from which all objects are being removed.

See also

Remove (theObject : RoseBase) : on page 138

RoseBase

Description

RoseBase is the root class of the RRTEI.

Public Operations

GetObject () : Object

Description

Retrieves the object's OLE interface object.

Note: This function is only valid for Rose Script; it has no meaning in Rose Automation.

Syntax

```
Set theOLEObject = theRoseBase.GetObject ( )
```

```
theOLEObject As Object
```

Returns the OLE automation interface object associated with the specified object.

```
theRoseBase As RoseRT.RoseBase
```

Instance of the object whose OLE interface object is being returned.

RRTEIObject

Description

Most elements in a Rose RealTime model derive, either directly or indirectly, from the RRTEIObject class. When you retrieve a model element as an object, you may not know what type of object you have retrieved.

Using RRTEIObject class operations, you can determine the type of the object.

Derived from RoseBase

Public Operations

IdentifyClass () : String

Description

Identifies the class of a Rose RealTime object

Note: For Rose RealTime Script, use the CanTypeCast method.

Syntax

```
theString = theRRTEIObject.IdentifyClass ( )
```

```
theString As String
```

Returns the RRTEIObject's class name.

```
ctheRRTEIObject As RoseRT. RRTEIObject
```

RRTEIObject whose class is being identified.

IsClass (theClassName : String) : Boolean

Description

Determines whether an object is a specified class.

Note: For Rose RealTime Script, use the CanTypeCast method.

Syntax

```
IsClass = theRRTEIObject.IsClass (theClassName)
```

```
IsClass As Boolean
```

Returns a value of True if its class matches the specified class name.

```
theRRTEIObject As RoseRT. RRTEIObject
```

RRTEIObject whose class is being checked.

```
theClassName As String
```

Name of the class for which the RRTEIObject is being checked.

RichTypes

RichTypes include

- RichType
  - Public Attributes

    Name : String on page 143

    Types : RichTypeValuesCollection on page 143

    Value : Integer on page 143
  - Public Operations

    GetObject () : Object on page 143
- *RichTypeValuesCollection* on page 144
  - Public Attributes

    Count : Integer on page 144

GetAt (id : Integer) : String on page 144

GetObject () : Object on page 145

RichType

Description

A rich type contains a set of values, of which only one is active at a time. They can be compared to a smart enumeration capable of being set using either the numeric or the string version of their values.

e.g.

ClassifierVisibilityKind' set of values are as follows:

```
(string version : numeric version)
"rsPublic" : 0
"rsProtected" : 1
"rsPrivate" : 2
"rsImplementation" : 3
```

A rich type derived class is always associated with an enumeration whose name is made of the rich type name (or substring of it) prefixed by “Rs”.

e.g.

ClassifierVisibilityKind rich type is associated with RsVisibilityKind enumeration.

The name of the enumeration's elements is made from the string version of the rich type value it represents.

e.g.

The ClassifierVisibilityKind rich type string value “rsPublic” is associated with the enumeration RsVisibilityKind's rsPublic element.

Here are valid ways to set a variable of type ClassifierVisibilityKind to public:

```
Set theClassifierVisibilityKind.Name = "rsPublic"
Set theClassifierVisibilityKind.Value = 0
Set theClassifierVisibilityKind.Value = rsPublic
```

To ease the use of rich types, the Value property is the default property of a rich type. This means that the Value property is assumed whenever a property or an operation is omitted while using a rich type.

e.g.

```
Set theClassifierVisibilityKind = 0
Set theClassifierVisibilityKind = rsPublic
```

Derived from RRTEIObject

Public Attributes

Name : String

Description

String version of the active value of the rich type.

Types : RichTypeValuesCollection

Description

Collection of the all the values that can be activated in the rich type, in string version.

Value : Integer

Description

Numeric version of the active value of the rich type.

Public Operations

GetObject () : Object

Description

Retrieves the object's OLE interface object.

Note: This operation is only valid for Rose RealTime Script; it has no meaning in Rose RealTime Automation.

Syntax

```
Set theOLEObject = theRichType.GetObject ( )
```

```
theOLEObject As Object
```

Returns the OLE automation interface object associated with the specified object.

```
theRichType As RoseRT.RichType
```

Instance of the rich type whose OLE interface object is being returned.

RichTypeValuesCollection

Description

Collection of all values that can be activated in a particular rich type.

Derived from RRTEIObject

Public Attributes

Count : Integer

Description

Number of values in the collection.

Public Operations

GetAt (id : Integer) : String

Description

Retrieves a particular value from the collection, given the value's position in the collection.

Syntax

```
Value = theRichTypeValuesCollection.GetAt ( theIndex )
```

```
Value As String
```

Returns the value from the collection.

```
theRichTypeValuesCollection As RichTypeValuesCollection
```

Collection from which to retrieve the value.

```
theIndex As Integer
```

Index (position) of the value in the collection. First value is at index 1.

GetObject () : Object

Description

Retrieves the object's OLE interface object.

Note: This function is only valid for Rose Script; it has no meaning in Rose Automation.

Syntax

```
Set theOLEObject = theRichTypeValuesCollection.GetObject ( )
```

```
theOLEObject As Object
```

Returns the OLE automation interface object associated with the specified object.

```
theRichTypeValuesCollection As RoseRT.RichTypeValuesCollection
```

Instance of the rich type values collection whose OLE interface object is being returned.

Model Classes

Model classes include

- *Component View Classes* on page 145

Component View Classes

Component View classes include

- *Component* on page 149
    - Public Attributes

AssignedClasses : ClassifierCollection on page 149

AssignedLogicalPackages : LogicalPackageCollection on page 149

CodeGenMakeDescription : String on page 149

CodeGenMakeFlags : String on page 149

CodeGenMakeName : String on page 150

CodeGenMakeOverridesFile : String on page 150

CodeGenMakeType : String on page 150

CompilationMakeDescription : String on page 150

CompilationMakeFlags : String on page 150

CompilationMakeName : String on page 150

CompilationMakeOverridesFile : String on page 150

CompilationMakeType : String on page 151

CompilerDescription : String on page 151

CompilerFlags : String on page 151

CompilerLibrary : String on page 151

CompilerName : String on page 151

DefaultArgs : String on page 152

Environment : String on page 152

ExecutableFileName : String on page 152

InclusionPaths : StringCollection on page 152

Inclusions : StringCollection on page 152

LinkerFlags : String on page 152

LinkerName : String on page 153

MultiThreaded : Boolean on page 153

OutputPath : String on page 153

ParentComponentPackage : ComponentPackage on page 153

Platform : String on page 153

RTSDescription : String on page 153

RTSType : String on page 154

TargetDescription : String on page 154

TargetLibrary : String on page 154

TopCapsule : Capsule on page 154

Type : String on page 155

UserLibraries : StringCollection on page 155

UserLibraryPaths : StringCollection on page 156

- Public Operations

AddComponentDependency (theDep : Component) : ComponentDependency on page 156

AddInclusion (inclusion : String) : Boolean on page 156

AddInclusionPath (pathName : String, ComputeDependencies : Boolean) : Boolean on page 157

AddRealizeRelation (theRelName : String, theInterfaceName : String) : RealizeRelation on page 158

AddUserLibrary (libraryName : String) : Boolean on page 158

AddUserLibraryPath (pathName : String) : Boolean on page 159

AssignClass (theClass : Classifier) : Boolean on page 159

AssignPackage (thePackage : LogicalPackage) : Boolean on page 160

Build (bUpdateAssignedClassList : Boolean) : Boolean

DeleteComponentDependency (theDep : ComponentDependency) : Boolean on page 161

DeleteInclusion (inclusion : String) : Boolean on page 161

DeleteInclusionPath (pathName : String) : Boolean on page 162

DeleteRealizeRelation (theRel : RealizeRelation) : Boolean on page 162

DeleteUserLibrary (libraryName : String) : Boolean on page 163

DeleteUserLibraryPath (pathName : String) : Boolean on page 163

Generate (bUpdateAssignedClassList : Boolean) : Boolean on page 164

GetAllClasses () : ClassifierCollection on page 165

GetComponentDependencies () : ComponentDependencyCollection on page 165

GetInclusionPathFlag (pathName : String) : Boolean on page 165

GetRealizeRelations () : RealizeRelationCollection on page 166

RebuildAll (bUpdateAssignedClassList : Boolean) : Boolean on page 166

RegenerateAll (bUpdateAssignedClassList : Boolean) : Boolean on page 167

ReverifyAll (bUpdateAssignedClassList : Boolean) : Boolean on page 168

UnassignClass (theClass : Classifier) : Boolean on page 168

UnassignPackage (thePackage : LogicalPackage) : Boolean on page 169

UpdateAssignedClassList () : Boolean on page 169

Verify (bUpdateAssignedClassList : Boolean) : Boolean on page 170

- *ComponentPackage* on page 170
  - Public Attributes

    ComponentDiagrams : ComponentDiagramCollection on page 170

    ComponentPackages : ComponentPackageCollection on page 171

    Components : ComponentCollection on page 171

    ParentComponentPackage : ComponentPackage on page 171
  - Public Operations

    AddComponent (theName : String) : Component on page 171

    AddComponentDiagram (name : String) : ComponentDiagram on page 172

    AddComponentPackage (theName : String) : ComponentPackage on page 172

    DeleteComponent (pIDispatch : Component) : Boolean on page 173

    DeleteComponentPackage (pIDispatch : ComponentPackage) : Boolean on page 173

    GetAllComponentPackages () : ComponentPackageCollection on page 174

    GetAllComponents () : ComponentCollection on page 174

    GetComponentDependencies () : ComponentDependencyCollection on page 175

    GetComponentPackageDependencies (theComponentPackage : ComponentPackage) : ComponentDependencyCollection on page 175

    GetVisibleComponentPackages () : ComponentPackageCollection on page 176

    RelocateComponent (theComponent : Component) : on page 176

RelocateComponentDiagram (theModDiagram : ComponentDiagram) : on page 177

RelocateComponentPackage (theComponentPackage : ComponentPackage) : on page 177

TopLevel () : Boolean on page 177

Component

Description

Components are used to model the physical elements that may reside on a node, such as executables, libraries, source files, documents. The component therefore represents the physical packaging of the logical elements, such as classes and capsules.

Derived from ModelElement

Public Attributes

AssignedClasses : ClassifierCollection

Description

Collection of classifiers assigned to a Component.

AssignedLogicalPackages : LogicalPackageCollection

Description

Collection of logical packages assigned to a Component.

CodeGenMakeDescription : String

Description

Used to describe any details regarding Code Generation Make configuration.

CodeGenMakeFlags : String

Description

Any flags supported to be passed to the make utility during Code Generation.

CodeGenMakeName : String

Description

The name of the make utility being used to control the code generation.

CodeGenMakeOverridesFile : String

Description

The overrides file is a makefile fragment which is included in the code generation makefile that allows for the addition of user-defined dependencies, compile, and link options in the code generation make files.

CodeGenMakeType : String

Description

Can be one of “Unix_make”, “Messmate” or “Gnu_make”.

CompilationMakeDescription : String

Description

Used to describe any details regarding Compilation Make configuration.

CompilationMakeFlags : String

Description

Any flags supported to be passed to the make utility during Compilation.

CompilationMakeName : String

Description

The name of the make utility being used to control the compilation and link of a component. The make name must be the exact name of the make command.

CompilationMakeOverridesFile : String

Description

The overrides file is a makefile fragment which is included in the compilation makefile that allows for the addition of user-defined dependencies, compile, and link options.

CompilationMakeType : String

Description

Can be one of “Unix_make”, “Messmate” or “Manlike”.

CompilerDescription : String

Description

Used to describe any details regarding Compiler configuration.

CompilerFlags : String

Description

Any flags supported by your compiler utility. This is where you would specify a parallel make flag to increase compilation efficiency.

CompilerLibrary : String

Description

Used to uniquely identify the Services Library set and build utilities that will be used to compile and link the component. The library name, which is actually a directory name of where to find the utilities and Services Library files, can be any legal directory name. However, in order to differentiate between the different variations of compiler and processors, a standard notation is commonly used. The compiler library name is composed of three parts: processor-compiler-version.

For example, the library name for an x86 processor built with version 6.0 of Microsoft Visual C++ would be called: x86-VisualC++-6.0

CompilerName : String

Description

Used to replace the pre-configured compiler shell command defined in libset.mk.

DefaultArgs : String

Description

Some platforms do not allow command line arguments to be passed to an executable at load time (namely, on some real-time operating systems). In this case, the default arguments provides a mechanism for getting execution arguments into the executable.

Note: The default arguments property will only be used for targets that cannot accept command line arguments. Targets that accept command line arguments will ignore the content of this property.

Environment : String

Description

Component build environment.

ExecutableFileName : String

Description

The name, or a name with an absolute path, of the executable that will be created as a result of the component being built.

InclusionPaths : StringCollection

Description

Collection of strings that represent the directory search set used by the compiler to find user-specified inclusion files. They are searched in the ordered specified in the collection.

Inclusions : StringCollection

Description

Component level inclusion files.

LinkerFlags : String

Description

Any flags supported by your linker utility.

LinkerName : String

Description

Used to replace the pre-configured linker shell command defined in libset.mk.

MultiThreaded : Boolean

Description

Indicates whether the component is compiled for a multi-threaded or single-threaded platform.

OutputPath : String

Description

The output path can be changed to allow you to set the directory into which the generated files resulting from a component build will be written. If left unspecified the generation and compilation results are stored in $ROSERT_HOME/[component name].

ParentComponentPackage : ComponentPackage

Description

Identifies the Component Package that contains the Component.

Platform : String

Description

The hardware on which you will run the executable, and hence identifies the platform for which to build the component. The target does not necessarily have to the same as the toolset is running on.

RTSDescription : String

Description

Used to describe any details regarding RTS configuration.

RTSType : String

Description

A pre-defined type that maps directly to a specific directory in the Rose RealTime installation directory. e.g. “C++ Target RTS”

TargetDescription : String

Description

Used to describe any details regarding Target configuration.

TargetLibrary : String

Description

Used to uniquely identify the Services Library set and build utilities that will be used to compile and link the component. The library name, which is actually a directory name of where to find the utilities and Services Library files, can be any legal directory name. However, in order to differentiate between the different variations of compiler and processors, a standard notation is commonly used. The compiler library name is composed of three parts: processor-compiler-version.

For example, the library name for an x86 processor built with version 6.0 of Microsoft Visual C++ would be called: x86-VisualC++-6.0

TopCapsule : Capsule

Description

Obsolete Property. This property is now implemented independently in each of the language add-ins if needed. Below is an example of how to address this in C++

```
Sub SetTopCapsule (theComponent As RoseRT.Component, theCapsule As
RoseRT.Capsule)
                ' First add the capsule as a reference if it isn't
                already
                If
                theComponent.AssignedClasses.FindFirst(theCapsule.Name
                ) = 0 Then
                   If Not theComponent.AssignClass(theCapsule) Then
```

```
            MsgBox "Error configuring component."
            Exit Sub
        End If
    End If

    toolName$ = "OT::CppExec"
    propertyName$ = "TopCapsule"

    If Not theComponent.OverrideProperty(toolName,
    propertyName, "") Then
        MsgBox "Error configuring component."
        Exit Sub
    End If

    Dim sp As RoseRT.StructuredProperty
    Set sp =
    theComponent.GetToolProperties(toolName).GetFirst(prop
    ertyName)
    sp.SetFieldValue "event_ui", "description",
    theCapsule.Name
    sp.SetFieldValue "event_ui", "caption", "Select..."

    Dim fullCapsuleName As String
    fullCapsuleName = """" + theCapsule.GetQualifiedName()
    + """" + " " + theCapsule.GetUniqueID()
    sp.SetFieldValue "", "", fullCapsuleName
End Sub
```

Type : String

Description

Component build type.

UserLibraries : StringCollection

Description

Any number of user libraries can be specified to be linked into an executable through user library items. The entry names themselves follow the convention associated with your compiler or operating system.

UserLibraryPaths : StringCollection

Description

Any number of entries can appear as library path items and as a group they comprise the directory search set used by the compiler to find user-specified libraries. They are searched in the order specified in the list (top to bottom).

Public Operations

AddComponentDependency (theDep : Component) : ComponentDependency

Description

Adds a Dependency relationship between two Components.

Syntax

```
Set theComponentDependency = theComponent.AddComponentDependency(
theDep )
```

`theComponentDependency As RoseRT.ComponentDependency`

Returns a new ComponentDependency whose dependent is theComponent and whose provider is theDep.

`theComponent As RoseRT.Component`

The ComponentDependency dependent component.

`theDep As String`

The ComponentDependency provider.

AddInclusion (inclusion : String) : Boolean

Description

Adds a component level inclusion file to be used by compiler.

Syntax

```
InclusionAdded = theComponent.AddInclusion( inclusion )
```

```
InclusionAdded As Boolean
```

Returns whether the new inclusion was added to theComponent.

```
theComponent As RoseRT.Component
```

The Component who gets a new inclusion added.

```
inclusion As String
```

The filename of the new inclusion file.

AddInclusionPath (pathName : String, ComputeDependencies : Boolean) : Boolean

Description

Adds a component level inclusion path to be used by the compiler.

Syntax

```
InclusionPathAdded = theComponent.AddInclusionPath( pathName,
ComputeDependencies )
```

```
InclusionPathAdded As Boolean
```

Returns a whether the new inclusion path was added to theComponent.

```
theComponent As RoseRT.Component
```

The Component who gets a new inclusion path added.

```
pathName As String
```

The pathname of the new inclusion path.

```
ComputeDependencies As Boolean
```

When set to True, the inclusion files in that directory are not considered during the dependency calculations.

AddRealizeRelation (theRelName : String, theInterfaceName : String) : RealizeRelation

Description

Adds a Realize relationship to a Component.

Syntax

```
Set theRealizeRel = theComponent.AddRealizeRel( theRelName,
theInterfaceName )
```

`theRealizeRel As RoseRT.RealizeRel`

Returns a new RealizeRelation whose client is theComponent and whose supplier is theInterfaceName.

`theComponent As RoseRT.Component`

The Component that realizes.

`theRelName As String`

The name of the new RealizeRelation.

`theInterfaceName As String`

The name of the supplier of the new RealizeRelation.

AddUserLibrary (libraryName : String) : Boolean

Description

Adds a component level library file to be used during builds.

Syntax

```
LibraryAdded = theComponent.AddUserLibrary( libraryName )
```

```
LibraryAdded As Boolean
```
Returns whether the new library was added to theComponent.

```
theComponent As RoseRT.Component
```
The Component who gets a new library added.

```
libraryName As String
```
The filename of the new library file.

AddUserLibraryPath (pathName : String) : Boolean

Description

Adds a component level library path to be used by during builds.

Syntax

```
LibraryPathAdded = theComponent.AddInclusionPath( pathName )
```

```
LibraryPathAdded As Boolean
```
Returns a whether the new library path was added to theComponent.
```
theComponent As RoseRT.Component
```
The Component who gets a new library path added.

```
pathName As String
```
The pathname of the new library path.

AssignClass (theClass : Classifier) : Boolean

Description

Assigns a classifier to a Component.

Syntax

```
ClassifierAssigned = theComponent.AssignClass( theClass )
```

```
ClassifierAssigned As Boolean
```

Returns whether theClass was assigned to theComponent.

```
theComponent As RoseRT.Component
```

The Component who gets assigned a theClass.

```
theClass As RoseRT.Classifier
```

Classifier to assign to theComponent.

AssignPackage (thePackage : LogicalPackage) : Boolean

Description

Assigns a package to a Component.

Syntax

```
PackageAssigned = theComponent.AssignPackage( thePackage )
```

```
PackageAssigned As Boolean
```

Returns whether thePackage was assigned to theComponent.

```
theComponent As RoseRT.Component
```

The Component who gets assigned thePackage.

```
thePackage As RoseRT.LogicalPackage
```

LogicalPackage to assign to theComponent.

Build (bUpdateAssignedClassList : Boolean) : Boolean

Description

Generates the source code for the component, and invokes the external compiler and linker to create an executable version of the component. Only the model elements that have changed will be generated and recompiled.

Syntax

```
BuildDone = theComponent.Build( bUpdateAssignedClassList )
```

```
BuildDone As Boolean
```

Returns whether Build operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets built.

```
bUpdateAssignedClassList As Boolean
```

Whether to update the assigned class list before performing the actual build.

DeleteComponentDependency (theDep : ComponentDependency) : Boolean

Description

Deletes a ComponentDependency relationship.

Syntax

```
ComponentDependencyDeleted = theComponent.DeleteComponentDependency(
theDep )
```

```
ComponentDependencyDeleted As Boolean
```

Returns whether theDep was deleted.

```
theComponent As RoseRT.Component
```

The Component to remove ComponentDependency from.

```
theDep As RoseRT.ComponentDependency
```

The ComponentDependency to remove from theComponent.

DeleteInclusion (inclusion : String) : Boolean

Description

Deletes an inclusion.

Syntax

```
InclusionDeleted = theComponent.DeleteInclusion( inclusion )
```

`InclusionDeleted As Boolean`

Returns whether inclusion was deleted.

`theComponent As RoseRT.Component`

The Component to remove inclusion from.

`inclusion As String`

The inclusion to remove from theComponent.

DeleteInclusionPath (pathName : String) : Boolean

Description

Deletes an inclusion path.

Syntax

```
InclusionPathDeleted = theComponent.DeleteInclusionPath( pathName )
```

`InclusionPathDeleted As Boolean`

Returns whether inclusion path was deleted.

`theComponent As RoseRT.Component`

The Component to remove inclusion path from.

`pathName As String`

The inclusion path to remove from theComponent.

DeleteRealizeRelation (theRel : RealizeRelation) : Boolean

Description

Deletes a realize relation.

Syntax

```
RealizeRelationDeleted = theComponent.DeleteRealizeRelation( theRel )
```

`RealizeRelationDeleted As Boolean`

Returns whether theRel Realize relation was deleted.

`theComponent As RoseRT.Component`

The Component to remove theRel from.

`theRel As RoseRT.RealizeRelation`

The Realize relation to remove from theComponent.

DeleteUserLibrary (libraryName : String) : Boolean

Description

Deletes a library.

Syntax

```
LibraryDeleted = theComponent.DeleteUserLibrary( libraryName )
```

`LibraryDeleted As Boolean`

Returns whether libraryName was deleted.

`theComponent As RoseRT.Component`

The Component to remove libraryName from.

`libraryName As String`

The library to remove from theComponent.

DeleteUserLibraryPath (pathName : String) : Boolean

Description

Deletes a library path.

Syntax

```
LibraryPathDeleted = theComponent.DeleteUserLibraryPath( pathName )
```

`LibraryPathDeleted As Boolean`

Returns whether library path was deleted.

`theComponent As RoseRT.Component`

The Component to remove library path from.

`pathName As String`

The library path to remove from theComponent.

Generate (bUpdateAssignedClassList : Boolean) : Boolean

Description

Generates the source code for the component but does not invoke the external compiler. Generation is incremental to previous build and generate requests. The Generate operation is usually used if the compilation is going to be invoked from outside the toolset.

Syntax

```
GenerationDone = theComponent.Generate( bUpdateAssignedClassList )
```

`GenerationDone As Boolean`

Returns whether Generation operation was performed.

`theComponent As RoseRT.Component`

The Component to generated code for.

`bUpdateAssignedClassList As Boolean`

Whether to update the assigned class list before performing the actual code generation.

GetAllClasses () : ClassifierCollection

Description

Returns all classifiers assigned to a Component.

Syntax

```
theClassifiers = theComponent.GetAllClasses()
```

```
theClassifiers As RoseRT.ClassifierCollection
```

Classifiers assigned to theComponent

```
theComponent As RoseRT.Component
```

The Component to return Classifiers assigned to.

GetComponentDependencies () : ComponentDependencyCollection

Description

Returns all ComponentDependency relations a Component is client of.

Syntax

```
theComponentDependencies = theComponent.GetComponentDependencies()
```

```
theComponentDependencies As RoseRT.ComponentDependencyCollection
```

ComponentDependencies of theComponent

```
theComponent As RoseRT.Component
```

The Component to return ComponentDependencies of.

GetInclusionPathFlag (pathName : String) : Boolean

Description

Returns the ComputeDependencies flag of an inclusion path of a Component.

Syntax

```
ComputeDependencies = theComponent.GetInclusionPathFlag( pathName )
```

```
ComputeDependencies As Boolean
```

Returns whether the ComputeDependencies flag is set for the pathname Inclusion Path.

```
theComponent As RoseRT.Component
```

The Component to that contains the Inclusion Path pathName.

```
pathName As String
```

Pathname of Inclusion Path to retrieve ComputeDependencies flag for.

GetRealizeRelations () : RealizeRelationCollection

Description

Returns all Realize relations of a Component.

Syntax

```
theRealizeRelations = theComponent.GetRealizeRelations()
```

```
theRealizeRelations As RoseRT.RealizeRelationCollection
```

Realize relations of theComponent

```
theComponent As RoseRT.Component
```

The Component to return Realize relations of.

RebuildAll (bUpdateAssignedClassList : Boolean) : Boolean

Description

Forces a complete build on a component. All classes referenced by the component will be regenerated, compiled, and linked.

Syntax

```
RebuildAllDone = theComponent.RebuildAll( bUpdateAssignedClassList )
```

```
RebuildAllDone As Boolean
```

Returns whether RebuildAll operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets rebuilt.

```
bUpdateAssignedClassList As Boolean
```

Whether to update the assigned class list before performing the actual RebuildAll.

RegenerateAll (bUpdateAssignedClassList : Boolean) : Boolean

Description

Initiates a model verification and generates the source code for the component but the external compiler is not invoked. Generation is not incremental to previous build and generate requests. The complete component is regenerated.

Syntax

```
RegeneratAllDone = theComponent.RegenerateAll(
bUpdateAssignedClassList )
```

```
RegeneratAllDone As Boolean
```

Returns whether RegenerateAll operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets regenerated.

```
bUpdateAssignedClassList As Boolean
```

Whether to update the assigned class list before performing the actual RegenerateAll.

ReverifyAll (bUpdateAssignedClassList : Boolean) : Boolean

Description

Run a complete verification of all elements. Normally, the toolset performs an incremental verification, checking only those elements that have changed since the last verify, and any elements affected by the changes. The reverify all command ignores the incremental changes and verifies the entire Component.

Syntax

```
ReverifyAllDone = theComponent.ReverifyAll( bUpdateAssignedClassList )
```

```
ReverifyAllDone As Boolean
```

Returns whether ReverifyAll operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets reverified.

```
bUpdateAssignedClassList As Boolean
```

Whether to update the assigned class list before performing the actual ReverifyAll.

UnassignClass (theClass : Classifier) : Boolean

Description

Unassigns a classifier from a Component.

Syntax

```
UnassignDone = theComponent.UnassignClass( theClass )
```

```
UnassignDone As Boolean
```

Returns whether Unassign operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets theClass unassigned from.

```
theClass As RoseRT.Classifier
```

The Classifier to unassign from the theComponent.

UnassignPackage (thePackage : LogicalPackage) : Boolean

Description

Unassigns a Logical Package from a Component.

Syntax

```
UnassignDone = theComponent.UnassignPackage( thePackage )
```

```
UnassignDone As Boolean
```

Returns whether Unassign operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets thePackage unassigned from.

```
thePackage As RoseRT.LogicalPackage
```

The Logical Package to unassign from the theComponent.

UpdateAssignedClassList () : Boolean

Description

Updates the assigned Classifier list of a Component based on the set of Classifiers referenced by the top Capsule or by any of its referenced Classifiers.

Syntax

```
UpdateDone = theComponent.UpdateAssignedClassList()
```

```
UpdateDone As Boolean
```

Returns whether Update operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets its classifier list updated.

Verify (bUpdateAssignedClassList : Boolean) : Boolean

Description

Initiate an internal check of the Component for consistency and errors. A Component verification is run every time a Component is either generated or built.

Syntax

```
VerifyDone = theComponent.Verify( bUpdateAssignedClassList )
```

```
VerifyDone As Boolean
```

Returns whether Verify operation was performed.

```
theComponent As RoseRT.Component
```

The Component who gets verified.

```
bUpdateAssignedClassList As Boolean
```

Whether to update the assigned class list before performing the actual Verify.

ComponentPackage

Description

A ComponentPackage is a collection of logically related components. (The ComponentPackage/component relationship is analogous to the logical package/class relationship).The ComponentPackage class exposes attributes and operations that allow you to define and manipulate ComponentPackages and their characteristics. Check the lists of attributes and operations for complete information.

Derived from Package

Public Attributes

ComponentDiagrams : ComponentDiagramCollection

Description

Contains the component diagrams belonging to the ComponentPackage.

ComponentPackages : ComponentPackageCollection

Description

Contains the ComponentPackages belonging to the ComponentPackage.

Components : ComponentCollection

Description

Contains the modules belonging to the subsystem.

ParentComponentPackage : ComponentPackage

Description

Identifies the ComponentPackage object that contains the ComponentPackage. If the ComponentPackage is the root ComponentPackage, then the value of parent ComponentPackage is set to Nothing.

Note: You can also use the TopLevel method to check for this condition.

Public Operations

AddComponent (theName : String) : Component

Description

Creates a new component in a ComponentPackage and returns it in the specified object.

Syntax

```
Set theComponent = theComponentPackage.AddComponent (theName)
```

```
theComponent As RoseRT.Component
```

Returns the newly created component object.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage to which new component is being added.

```
theName As String
```

Name of the component to be created.

AddComponentDiagram (name : String) : ComponentDiagram

Description

Creates a new component diagram in a ComponentPackage and returns it in the specified object.

Syntax

```
Set theComponentDiagram = theComponentPackage.AddComponentDiagram
(theName)
```

```
theComponentDiagram As RoseRT.ComponentDiagram
```

Returns the newly created component diagram object.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage to which new component diagram is being added.

```
theName As String
```

Name of the component diagram to be created.

AddComponentPackage (theName : String) : ComponentPackage

Description

Creates a new ComponentPackage in a model and returns it in the specified ComponentPackage object.

Syntax

```
Set theComponentPackage = theObject.AddComponentPackage (theName)
```

```
theComponentPackage As RoseRT.ComponentPackage
```

Returns the newly created ComponentPackage.

```
theObject As RoseRT.ComponentPackage
```

Instance of the ComponentPackage being created.

```
theName As String
```

Name of the ComponentPackage being created.

DeleteComponent (pIDispatch : Component) : Boolean

Description

Deletes a component from a ComponentPackage.

Syntax

```
IsDeleted = theComponentPackage.DeleteComponent (theComponent)
```

```
IsDeleted As Boolean
```

Returns a value of True when the component is successfully deleted.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage from which to delete the module.

```
theComponent As RoseRT.Component
```

Component being deleted.

DeleteComponentPackage (pIDispatch : ComponentPackage) : Boolean

Description

Deletes a ComponentPackage from a ComponentPackage.

Syntax

```
IsDeleted = theComponentPackage.DeleteComponentPackage
(theComponentPackage)
```

```
IsDeleted As Boolean
```

Returns a value of True when the ComponentPackage is successfully deleted.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage from which to delete the ComponentPackage.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage being deleted.

GetAllComponentPackages () : ComponentPackageCollection

Description

Retrieves all ComponentPackages belonging to a ComponentPackage.

Syntax

```
Set theComponentPackages = theComponentPackage.GetAllComponentPackages
( )
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns all ComponentPackage belonging to the ComponentPackage.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage whose ComponentPackages are being retrieved.

GetAllComponents () : ComponentCollection

Description

Retrieves all components belonging to a ComponentPackage.

Syntax

```
Set theComponents = theComponentPackage.GetAllComponents ( )
```

```
theComponents As RoseRT.ComponentCollection
```

Returns all components belonging to the ComponentPackage.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage whose components are being retrieved.

GetComponentDependencies () : ComponentDependencyCollection

Description

Returns all ComponentDependency relations a ComponentPackage is client of.

Syntax

```
theComponentDependencies =
theComponentPackage.GetComponentDependencies()
```

```
theComponentDependencies As RoseRT.ComponentDependencyCollection
```

ComponentDependencies theComponentPackage is client of.

```
theComponentPackage As RoseRT.ComponentPackage
```

The ComponentPackage to the ComponentDependencies it is client of.

GetComponentPackageDependencies (theComponentPackage : ComponentPackage) : ComponentDependencyCollection

Description

Retrieves the ComponentDependency collection owned by a ComponentPackage whose supplier is another specified ComponentPackage. The clients of these relations are Components.

Syntax

```
Set theComponentDependencies =
theComponentPackage.GetComponentPackageDependencies(
theSupplierComponentPackage )
```

```
theComponentDependencies As RoseRT.ComponentDependencyCollection
```

Returns the component dependency collection owned by the theComponentPackage whose supplier is theSupplierComponentPackage.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage that owns the collection of ComponentDependency being retrieved.

```
theSupplierComponentPackage As RoseRT.ComponentPackage
```

Supplier of the component dependencies retrieved.

GetVisibleComponentPackages () : ComponentPackageCollection

Description

Retrieves all ComponentPackages that are visible from a ComponentPackage. This includes ComponentPackage containing Component that are visible from the queried Component Package.

Syntax

```
Set theComponentPackages =
theComponentPackage.GetVisibleComponentPackages ( )
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns all ComponentPackage visible from the ComponentPackage.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage whose visible ComponentPackages are being retrieved.

RelocateComponent (theComponent : Component) :

Description

Relocates a component in a ComponentPackage.

Syntax

```
theComponentPackage.RelocateComponent theComponent
```

```
theComponentPackage As RoseRT.ComponentPackage
```

The component package to relocate a component into.

```
theComponent As RoseRT.Component
```

The component to relocate.

RelocateComponentDiagram (theModDiagram : ComponentDiagram) :

Description

Relocates a component diagram in a ComponentPackage.

Syntax

```
theComponentPackage.RelocateComponentDiagram theComponentDiagram
```

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage that contains the component diagram being relocated.

```
theComponentDiagram As RoseRT.ComponentDiagram
```

Component diagram being relocated.

RelocateComponentPackage (theComponentPackage : ComponentPackage) :

Description

Relocates a ComponentPackage in a model.

Syntax

```
theComponentPackage.RelocateComponentPackage theComponentPackage
```

```
theComponentPackage As RoseRT.ComponentPackage
```

Component package that contains the ComponentPackage being relocated.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage being relocated.

TopLevel () : Boolean

Description

Determines whether the specified object is the root ComponentPackage.

Syntax

```
IsTopLevel = theComponentPackage.TopLevel ( )
IsTopLevel As Boolean
```

Returns a value of True if the specified object is the root component package.

`theComponentPackage As RoseRT.ComponentPackage`

ComponentPackage object being tested as root ComponentPackage.

Core Model Classes

Core Model classes include

- *ControllableElement* on page 184
    - Public Attributes

      ControlNewUnits : Boolean on page 184
    - Public Operations

      Control () : Boolean on page 185

      ControlChildElements (Recursive : Boolean) : Boolean on page 185

      ControlTo (Path : String) : Boolean on page 185

      GetChildDirName () : String on page 186

      GetContainingControlledElement () : ControllableElement on page 186

      GetControlledChildElements (bRecursive : Boolean) : ControllableElementCollection on page 187

      GetFileName () : String on page 187

      GetVersion () : String on page 188

      IsCheckedOut () : Boolean on page 188

      IsChildDirCheckedOut () : Boolean on page 189

      IsChildDirUnderSourceControl () : Boolean on page 189

      IsControllableElementContainer () : Boolean on page 190

      IsControlled () : Boolean on page 190

      IsLoaded () : Boolean on page 191

      IsModifiable () : Boolean on page 191

IsModified () : Boolean on page 192

IsOwned () : Boolean on page 192

IsUnderSourceControl () : Boolean on page 192

Save () : Boolean on page 193

Uncontrol () : Boolean on page 193

UncontrolChildElements (Recursive : Boolean) : Boolean on page 194

- *DefaultModelProperties* on page 194
  - Public Operations

    AddDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String, PropType : String, Value : String) : Boolean on page 195

    CloneDefaultPropertySet (ClassName : String, ToolName : String, ExistingSetName : String, NewSetName : String) : Boolean on page 196

    CreateDefaultPropertySet (ClassName : String, ToolName : String, NewSetName : String) : Boolean on page 197

    DeleteDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String) : Boolean on page 198

    DeleteDefaultPropertySet (ClassName : String, ToolName : String, SetName : String) : Boolean on page 199

    FindDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String) : Property on page 200

    GetDefaultPropertySet (ClassName : String, ToolName : String, SetName : String) : PropertyCollection on page 201

    GetDefaultSetNames (ClassName : String, ToolName : String) : StringCollection on page 201

    GetToolNames (Parameter1 : String) : StringCollection on page 202

    IsToolVisible (theToolName : String) : Boolean on page 203

    SetToolVisibility (theToolName : String, Visibility : Boolean) : on page 203

- *Element* on page 204
  - Public Attributes

    Application : Application on page 204

    Model : Model on page 204

    Name : String on page 205
  - Public Operations

    CreateProperty (theToolName : String, thePropName : String, theValue : String, theType : String) : Boolean on page 205

    FindDefaultProperty (theToolName : String, thePropName : String) : Property on page 206

    FindProperty (theToolName : String, thePropName : String) : Property on page 206

    GetAllProperties () : PropertyCollection on page 207

    GetCurrentPropertySetName (ToolName : String) : String on page 207

    GetDefaultPropertyValue (theToolName : String, thePropName : String) : String on page 208

    GetDefaultSetNames (ToolName : String) : StringCollection on page 208

    GetPropertyClassName () : String on page 209

    GetPropertyValue (theToolName : String, thePropName : String) : String on page 209

    GetQualifiedName () : String on page 210

    GetToolNames () : StringCollection on page 211

    GetToolProperties (theToolName : String) : PropertyCollection on page 211

    GetUniqueID () : String on page 211

    InheritProperty (theToolName : String, thePropName : String) : Boolean on page 212

    IsDefaultProperty (theToolName : String, thePropName : String) : Boolean on page 213

    IsOverriddenProperty (theToolName : String, thePropName : String) : Boolean on page 213

OverrideProperty (theToolName : String, thePropName : String, theValue : String) : Boolean on page 214

SetCurrentPropertySetName (ToolName : String, SetName : String) : Boolean on page 215

- *ExternalDocument* on page 215
  - Public Attributes

    ParentLogicalPackage : LogicalPackage on page 216

    Path : String on page 216

    URL : String on page 216
  - Public Operations

    IsURL () : Boolean on page 216

    Open (szAppPath : String) : Boolean on page 217
- *Model* on page 218
  - Public Attributes

    ActiveComponent : Component on page 218

    DefaultProperties : DefaultModelProperties on page 218

    DeploymentDiagram : DeploymentDiagram on page 218

    RootComponentPackage : ComponentPackage on page 219

    RootDeploymentPackage : DeploymentPackage on page 219

    RootLogicalPackage : LogicalPackage on page 219

    RootUseCaseLogicalPackage : LogicalPackage on page 219

    UseCases : UseCaseCollection on page 219
  - Public Operations

    AddActiveComponentInstance (ComponentInstanceToAdd : ComponentInstance) : Boolean on page 219

    AddDevice (pName : String) : Device on page 220

    ControlAllUnits (bControlAllUnits : Boolean) : Boolean on page 221

    DeleteDevice (pDevice : Device) : Boolean on page 221

    DeleteProcessor (pProcessor : Processor) : Boolean on page 222

FindCapsuleWithID (UniqueID : String) : Capsule on page 222

FindCapsules (CapsuleName : String) : CapsuleCollection on page 223

FindClassWithID (UniqueID : String) : Class on page 223

FindClasses (ClassName : String) : ClassCollection on page 224

FindLogicalPackageWithID (UniqueID : String) : LogicalPackage on page 224

FindLogicalPackages (LogicalPackageName : String) : LogicalPackageCollection on page 225

FindModelElementWithID (UniqueID : String) : ModelElement on page 225

FindModelElements (ModelElementName : String) : ModelElementCollection on page 226

FindProtocolWithID (UniqueID : String) : Protocol on page 226

FindProtocols (ProtocolName : String) : ProtocolCollection on page 227

GetActiveComponentInstances () : ComponentInstanceCollection on page 227

GetActiveDiagram () : Diagram on page 228

GetAllAssociations () : AssociationCollection on page 228

GetAllCapsules () : CapsuleCollection on page 229

GetAllClasses () : ClassCollection on page 229

GetAllComponentPackages () : ComponentPackageCollection on page 230

GetAllComponents () : ComponentCollection on page 230

GetAllDevices () : DeviceCollection on page 230

GetAllLogicalPackages () : LogicalPackageCollection on page 231

GetAllProcessors () : ProcessorCollection on page 231

GetAllProtocols () : ProtocolCollection on page 232

GetAllUseCases () : UseCaseCollection on page 232

GetSelectedCapsules () : CapsuleCollection on page 232

GetSelectedClasses () : ClassCollection on page 233

GetSelectedComponentPackages () : ComponentPackageCollection on page 233

GetSelectedComponents () : ComponentCollection on page 234

GetSelectedLogicalPackages () : LogicalPackageCollection on page 234

GetSelectedModelElements () : ModelElementCollection on page 234

GetSelectedProtocols () : ProtocolCollection on page 235

GetSelectedUseCases () : UseCaseCollection on page 235

RemoveActiveComponentInstance (ComponentInstanceToRemove : ComponentInstance) : Boolean on page 236

- *ModelElement* on page 236
  - Public Attributes

    Documentation : String on page 237

    ExternalDocuments : ExternalDocumentCollection on page 237

    LocalizedStereotype : String on page 237

    Stereotype : String on page 237
  - Public Operations

    AddExternalDocument (szName : String, iType : RsExternalDocumentType) : ExternalDocument on page 237

    DeleteExternalDocument (pIDispatch : ExternalDocument) : Boolean on page 238

    GetModelElement () : ModelElement on page 238

    OpenSpecification () : Boolean on page 239
- *Package* on page 239
  - Public Operations

    AddSharedUnit (FileName : String) : Boolean on page 240

    AddUnit (FileName : String) : Boolean on page 240

    ImportFile (FileName : String) : Boolean on page 241

    ImportFileEx (FileName : String) : ControllableElementCollection on page 241

    IsRootPackage () : Boolean on page 242

    TopLevel () : Boolean on page 242

        - *Property* on page 243
    - Public Attributes

      Name : String on page 243

      ToolName : String on page 243

      Type : String on page 243

      Value : String on page 244
- *RsExternalDocumentType* on page 244
    - Public Attributes

      rsFile : Integer = 1 on page 244

      rsURL : Integer = 2 on page 244
- *StructuredProperty* on page 244
    - Public Operations

      GetFieldValue on page 245

      SetFieldValue on page 246

ControllableElement

Description

The ControllableElement class is an abstract class that exposes Rational Rose RealTime unit functionality in the RRTEI. ControllableElements are either controlled, or contained in a controlled ControllableElement. A controlled ControllableElement has an associated file where it stores its persistent state and the one of its contained ControllableElements.

Derived from Element

Public Attributes

ControlNewUnits : Boolean

Description

Determines whether new child units will be created as controlled units.

Public Operations

Control () : Boolean

Description

Controls a ControllableElement in default unit file.

Syntax

```
IsControlled = theControllableElement.Control()
```

```
IsControlled As Boolean
```

Whether theControllableElement is controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to control.

ControlChildElements (Recursive : Boolean) : Boolean

Description

Controls all children of a ControllableElement.

Syntax

```
AreControlled = theControllableElement.ControlChildElements( Recursive
As Boolean )
```

```
AreControlled As Boolean
```

Whether all controllable children of theControllableElement are controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to control children of.

```
Recursive As Boolean
```

Specifies whether to control children's children units two.

ControlTo (Path : String) : Boolean

Description

Controls a ControllableElement.

Syntax

```
IsControlled = theControllableElement.ControlTo( Path As String )
```

`IsControlled As Boolean`

Whether theControllableElement is controlled.

`theControllableElement As RoseRT.ControllableElement`

The Controllable Element to control.

`Path As String`

Pathname of controlled element.

GetChildDirName () : String

Description

Returns the directory name of the folder containing the persistent state of a controlled ControllableElement's children controllable elements.

Syntax

```
theDirectoryName = theControllableElement.GetChildDirName()
```

`theDirectoryName As String`

The directory name where theControllableElement's children controllable elements are stores. Notice that an empty string is returned if theControllableElement is NOT controlled or if it can not contain children Controllable Elements.

`theControllableElement As RoseRT.ControllableElement`

The Controllable Element to retrieve the directory name used to store that Controllable Element's children.

GetContainingControlledElement () : ControllableElement

Description

Returns the ControllableElement that controls a ControllableElement. May return self.

Syntax

```
theContainingControlledElement =
theControllableElement.GetContainingControlledElement()
```

```
theContainingControlledElement As RoseRT.ControllableElement
```

The ControllableElement that controls theControllableElement

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to get the controlled ControllableElement it is contained in.

GetControlledChildElements (bRecursive : Boolean) : ControllableElementCollection

Description

Returns the collection of ControllableElement contained in a ControllableElement.

Syntax

```
theChildControlledElements =
theControllableElement.GetControlledChildElements( bRecursive )
```

```
theChildControlledElements As RoseRT.ControllableElementCollection
```

The ControllableElement that controls theControllableElement

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to get the controlled ControllableElement it is contained in.

```
bRecursive As Boolean
```

Whether get the child ControllableElement's child recursively.

GetFileName () : String

Description

Returns the fully qualified name of the file containing the persistent state to a controlled ControllableElement and its children.

Syntax

```
theFileName = theControllableElement.GetFileName()
```

```
theFileName As String
```

The fully qualified name of theControllableElement's unit file. Notice that an empty string is returned if theControllableElement is NOT controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the unit fully qualified filename from.

GetVersion () : String

Description

Returns the Source Control version associated with a controlled ControllableElement.

Syntax

```
theVersion = theControllableElement.GetVersion()
```

```
theVersion As String
```

The Source Control version of theControllableElement. Notice that an empty string is returned if theControllableElement is NOT controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the Source Control version from.

IsCheckedOut () : Boolean

Description

Returns whether a controlled ControllableElement is checked out of Source Control.

Syntax

```
IsCheckedOut = theControllableElement.IsCheckedOut()
```

```
IsCheckedOut As Boolean
```

Whether theControllableElement is checked out from Source Control. Notice that False is always returned if theControllableElement is NOT controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve Source Control checkout status from.

IsChildDirCheckedOut () : Boolean

Description

Returns whether a controlled ControllableElement's child controllable elements' directory is checked out of Source Control.

Syntax

```
IsChildDirCheckedOut = theControllableElement.IsCheckedOut()
```

```
IsChildDirCheckedOut As Boolean
```

Whether theControllableElement's child controllable elements' directory is checked out from Source Control. Notice that False is always returned if theControllableElement is NOT controlled. Controllable Element that can not contain children Controllable Elements always return False.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element whose child controllable elements' directory is used to retrieve Source Control checkout status from.

IsChildDirUnderSourceControl () : Boolean

Description

Returns whether a controlled ControllableElement's child controllable elements' directory is under Source Control.

Syntax

```
IsChildDirUserSourceControl =
theControllableElement.IsChildDirUserSourceControl()
```

```
IsChildDirUserSourceControl As Boolean
```

Whether child directory of theControllableElement is under SourceControl. Non Controlled ControllableElement always return False. Controllable Element that can not contain children Controllable Elements always return False.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the IsChildDirUserSourceControl status from.

IsControllableElementContainer () : Boolean

Description

Returns whether the Controllable Element can contain child Controllable Elements.

Syntax

```
IsControllableElementContainer =
theControllableElement.IsControllableElementContainer()
```

```
IsControllableElementContainer As Boolean
```

Whether theControllableElement can contain child Controllable Elements.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve whether it can contain child Controllable Element.

IsControlled () : Boolean

Description

Returns whether a ControllableElement is controlled.

Syntax

```
IsControlled = theControllableElement.IsControlled()
```

```
IsControlled As Boolean
```

Whether theControllableElement is controlled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the Controlled status from.

IsLoaded () : Boolean

Description

Returns whether a Controlled ControllableElement is Loaded. A controlled ControllableElement is always in the Loaded state except in very rare situations.

Syntax

```
IsLoaded = theControllableElement.IsLoaded()
```

```
IsLoaded As Boolean
```

Whether theControllableElement is loaded. Notice that a non controlled Controllable Element will always return False.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the Loaded status from.

IsModifiable () : Boolean

Description

Returns whether a ControllableElement is modifiable.

Syntax

```
IsModifiable = theControllableElement.IsModifiable()
```

```
IsModifiable As Boolean
```

Whether theControllableElement can be modified. Notice that a non controlled Controllable Element will always base its ModifiableState on the one of its Containing ControllableElement.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the Modifiable status from.

IsModified () : Boolean

Description

Returns whether the ControllableElement's ContainingControllableElement, or its children have been modified.

Syntax

```
IsModified = theControllableElement.IsModified()
```

```
IsModified As Boolean
```

Whether theControllableElement's ContainingControllableElemtn or its children has been modified since last save.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the Modified status from.

IsOwned () : Boolean

Description

Returns whether a ControllableElement is owned by the Model.

Syntax

```
IsOwned = theControllableElement.IsOwned()
```

```
IsOwned As Boolean
```

Whether theControllableElement IsOwned by the Model. The RTSClasses logical package is an example of a ControllableElement not owned by the model.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the IsOwned status from.

IsUnderSourceControl () : Boolean

Description

Returns whether a controlled ControllableElement is under Source Control.

Syntax

```
IsUserSourceControl = theControllableElement.IsUserSourceControl()
```

```
IsUserSourceControl As Boolean
```

Whether theControllableElement is under SourceControl. Non Controlled ControllableElement always return False.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to retrieve the IsUnderSourceControl status from.

Save () : Boolean

Description

Saves a controlled ControllableElement.

Syntax

```
Saved = theControllableElement.Save()
```

```
Saved As Boolean
```

Whether theControllableElement was saved. Non Controlled ControllableElement always return False.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to save.

Uncontrol () : Boolean

Description

Uncontrols a ControllableElement.

Syntax

```
IsUncontrolled = theControllableElement.Control()
```

```
IsUncontrolled As Boolean
```

Whether theControllableElement is uncontrolled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to uncontrol.

UncontrolChildElements (Recursive : Boolean) : Boolean

Description

Uncontrols all children of a ControllableElement.

Syntax

```
AreUncontrolled = theControllableElement.UncontrolChildElements(
Recursive As Boolean )
```

```
AreUncontrolled As Boolean
```

Whether all controllable children of theControllableElement are uncontrolled.

```
theControllableElement As RoseRT.ControllableElement
```

The Controllable Element to uncontrol children of.

```
Recursive As Boolean
```

Specifies whether to uncontrol children's children units two.

DefaultModelProperties

Description

The DefaultModelProperties Class is a container for the default model properties that belong to a model. There is one and only one DefaultModelProperties object per model.

Note: If you use PropertyCollection methods to retrieve model properties, the collection can include both default and non-default model properties.

Derived from ModelElement

Public Operations

AddDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String, PropType : String, Value : String) : Boolean

Description

Adds a default property to a model:

- The class name, tool name and set name determine where the property is added.
- The property name, property type, and property type define the property itself.

Syntax

```
IsAdded = theProperties.AddDefaultProperty (theClassName, theToolName,
theSetName, thePropName, thePropType, theValue)
```

`IsAdded As Boolean`

Returns a value of True when the default property is successfully added.

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theClassName As String`

Name of the class to which the default property applies; corresponds to the Type field in the property specification editor of the Rose user interface. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

`theToolName As String`

Name of the tool to which the default property applies; If the tool does not exist, it will be created.

`theSetName As String`

Name of the property set to which the default property applies.

`thePropName As String`

Name of the default property.

`thePropType As String`

PropertyType of the default property.

`theValue As String`

Value of the default property.

See also

AddDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String, PropType : String, Value : String) : Boolean on page 195

CloneDefaultPropertySet (ClassName : String, ToolName : String, ExistingSetName : String, NewSetName : String) : Boolean

Description

Creates a new default property set by cloning an existing property set.

Syntax

```
IsCloned = theProperties.CloneDefaultPropertySet (theClassName,
theToolName, theExistingSetName, theNewSetName)
```

`IsCloned As Boolean`

Returns a value of True when the default property set is successfully cloned.

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theClassName As String`

Name of the extensibility class to which the new default property set applies. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

`theToolName As String`

Name of the tool to which the new default property set applies.

`theExistingSetName As String`

Name of the existing default property set being cloned.

`theNewSetName As String`

Name of the new default property set created from the clone.

See also

CreateDefaultPropertySet (ClassName : String, ToolName : String, NewSetName : String) : Boolean on page 197

CreateDefaultPropertySet (ClassName : String, ToolName : String, NewSetName : String) : Boolean

Description

Creates a new default property set without using an existing property set as a base.

Syntax

```
IsCreated = theProperties.CreateDefaultPropertySet (theClassName,
theToolName, theNewSetName)
```

`IsCreated As Boolean`

Returns a value of True when the default property set is successfully created.

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theClassName As String`

Name of the extensibility class to which the new default property set applies. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

```
theToolName As String
```

Name of the tool to which the new default property set applies.

```
theNewSetName As String
```

Name of the newly created default property set.

See also

CloneDefaultPropertySet (ClassName : String, ToolName : String, ExistingSetName : String, NewSetName : String) : Boolean on page 196

DeleteDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String) : Boolean

Description

Deletes a default property from a model. This method only deletes the property that belongs to the given class, tool, and set. If a different combination of class, tool, and set contains a default property with the same property name, that default property will remain intact and will not be deleted.

Syntax

```
IsDeleted = theProperties.DeleteDefaultProperty (theClassName,
theToolName, theSetName, thePropName)
```

```
IsDeleted As Boolean
```

Returns a value of True when the default property is successfully deleted.

```
theProperties As RoseRT.DefaultModelProperties
```

Contains the default properties belonging to the model.

```
theClassName As String
```

Name of the extensibility class to which the default property applies. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

```
theToolName As String
```

Name of the tool to which the default property applies.

`theSetName As String`

Name of the property set to which the default property applies.

`thePropName As String`

Name of the default property to delete.

DeleteDefaultPropertySet (ClassName : String, ToolName : String, SetName : String) : Boolean

Description

Deletes a default property set from a model.

Syntax

```
IsDeleted = theProperties.DeleteDefaultPropertySet (theClassName,
theToolName, theSetName)
```

`IsDeleted As Boolean`

Returns a value of True when the default property set is successfully deleted.

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theClassName As String`

Name of the extensibility class to which the deleted default property set applies. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

`theToolName As String`

Name of the tool to which the deleted default property set applies.

`theSetName As String`

Name of the default property set to delete.

FindDefaultProperty (ClassName : String, ToolName : String, SetName : String, PropName : String) : Property

Description

Finds a specific default model property, given the name of the class, tool, and property set that contain it.

Syntax

```
theProperty = theProperties.FindDefaultProperty (theClassName,
theToolName, theSetName, thePropName)
```

`theProperty As RoseRT.Property`

Returns the default model property, if found. Returns an empty value if the property does not exist.

`theProperties As RoseRT.DefaultModelProperties`

Contains the properties belonging to the model .

`theClassName As String`

Name of the extensibility class to search Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

`theToolName As String`

Name of the tool to search.

`theSetName As String`

Name of the default property set to search.

`thePropName As String`

Name of the default property to find.

GetDefaultPropertySet (ClassName : String, ToolName : String, SetName : String) : PropertyCollection

Description

Retrieves the set of default model properties that belongs to a given extensibility class and tool.

Syntax

```
Set theSet = theProperties.GetDefaultPropertySet (theClassName,
theToolName)
```

`theSet As DefaultModelProperties`

Returns the set of default model properties that belongs to the specified extensibility class and tool.

`theProperties As RoseRT.DefaultModelProperties`

Contains the properties belonging to the model.

`theClassName As String`

Name of the extensibility class to which the retrieved default property set belongs. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

`theToolName As String`

Name of the tool to which the retrieved default property set belongs.

GetDefaultSetNames (ClassName : String, ToolName : String) : StringCollection

Description

Retrieves the names of the default property sets that contain the model's default properties.

Syntax

```
theSetNames = theProperties.GetDefaultSetNames (theClassName,
theToolName)
```

```
theSetNames As StringCollection
```

Returns a StringCollection containing the valid default property set names for the given extensibility class and tool.

```
theProperties As RoseRT.DefaultModelProperties
```

Contains the default properties belonging to the model.

```
theClassName As String
```

Name of the extensibility class for which you are retrieving valid default property set names. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

```
theToolName As String
```

Name of the tool for which you are retrieving valid default property set names.

GetToolNames (Parameter1 : String) : StringCollection

Description

Retrieves the names of the tools associated with the given properties and class name.

Syntax

```
Set theToolNames = theProperties.GetToolNames (theClassName)
```

```
theToolNames As RoseRT.StringCollection
```

Returns a StringCollection containing the valid tool names for the given extensibility class.

```
theProperties As RoseRT.DefaultModelProperties
```

Contains the default properties belonging to the model.

```
theClassName As String
```

Name of the extensibility class for which you are retrieving valid tool names. Use the Element.GetPropertyClassName method to retrieve the valid string to pass as theClassName for a model element.

IsToolVisible (theToolName : String) : Boolean

Description

Determines whether the property tab for the given tool will appear in the property specification.

Syntax

```
IsVisible = theProperties.IsToolVisible (theToolName)
```

`IsVisible As Boolean`

Returns a value of True if the default model properties' tool is visible.

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theToolName As String`

Name of the tool to which the default properties belong.

SetToolVisibility (theToolName : String, Visibility : Boolean) :

Description

Sets the tool's visibility; that is, whether the property tab for the given tool will appear in the property specification.

Syntax

```
theProperties.SetToolVisibility theToolName, Visibility
```

`theProperties As RoseRT.DefaultModelProperties`

Contains the default properties belonging to the model.

`theToolName As String`

Name of the tool whose visibility is being set.

```
Visibility As Boolean
```

Set to True to make the tool visible; set to False to make the tool invisible.

Element

Description

The element class provides the interface to model properties.

Every object in a Rose RealTime model (including the model itself) is an element. And every element in a Rose RealTime model has a name and /or a unique ID. Following this logic, you can use Element Class methods to obtain the ID for any item in the current model, and from there get or set its properties and property sets.

The unique element ID also provides the most direct means of accessing an item from a collection. While you can still use GetFirst and GetNext methods to iterate through a collection, you can also use the GetwithUniqueID method to obtain the item right away, without searching through the collection.

Derived from RRTElObject

Public Attributes

Application : Application

Description

Name of a model element

Model : Model

Description

Name of a model element

Name : String

Description

Name of a model element

Public Operations

CreateProperty (theToolName : String, thePropName : String, theValue : String, theType : String) : Boolean

Description

Creates a new property for a given model element and tool.

Syntax

```
IsCreated = theElement.CreateProperty (theToolName, thePropName,
theValue, theType)
```

`IsCreated As Boolean`

Returns a value of True when the property is created for the element.

`theElement As RoseRT.Element`

Element for which the property is being created.

`theToolName As String`

Name of the tool to which the property applies.

`thePropName As String`

Name of the property being created.

`theValue As String`

Default value of the new property.

`theType As String`

Property type of the property.

FindDefaultProperty (theToolName : String, thePropName : String) : Property

Description

Returns the default property given the tool name and property name.

Syntax

```
Set theProperty = theElement.FindDefaultProperty (theToolName,
thePropName)
```

```
theProperty As RoseRT.Property
```

Returns the default property given its name and associated tool name.

```
theElement As RoseRT.Element
```

Model element whose default property is being returned.

```
theToolName As String
```

Name of the tool to which the default property applies.

```
thePropName As String
```

Name of the property being retrieved.

FindProperty (theToolName : String, thePropName : String) : Property

Description

Returns the property given the tool name and property names.

Syntax

```
Set theProperty = theElement.FindDefaultProperty (theToolName,
thePropName)
```

```
theProperty As RoseRT.Property
```

Returns the property given its name and its associated tool name.

```
theElement As RoseRT.Element
```

Model element whose property is being returned.

```
theToolName As String
```

Name of the tool to which the property applies.

```
thePropName As String
```

Name of the property to return.

GetAllProperties () : PropertyCollection

Description

Returns the collection of properties belonging to the specified element

Syntax

```
Set theProperties = theElement.GetAllProperties ()
```

```
theProperties As RoseRT.PropertyCollection
```

Returns the collection of properties belonging to the specified element.

```
theElement As RoseRT.Element
```

Model element whose properties are being returned.

GetCurrentPropertySetName (ToolName : String) : String

Description

Returns the name of the currently active property set given the element and a tool name.

Syntax

```
theName = theElement.GetCurrentPropertySetName (theToolName)
```

```
theName As String
```

Returns the name of the currently active property set.

```
theElement As RoseRT.Element
```

Element to which the property set belongs.

```
theToolName As String
```

Name of the tool to which the property set belongs.

GetDefaultPropertyValue (theToolName : String, thePropName : String) : String

Description

Retrieves the default property value given a tool name and property name.

Syntax

```
theValue = theElement.FindDefaultProperty (theToolName, thePropName)
```

```
theValue  As String
```

Returns the default property value for the specified tool name and property name.

```
theElement As RoseRT.Element
```

Element for which the default property value is being retrieved.

```
theToolName As String
```

Name of the tool to which the property applies.

```
thePropName As String
```

Name of the property being retrieved.

GetDefaultSetNames (ToolName : String) : StringCollection

Description

Retrieves the names of the default property sets defined for the specified element and tool.

Syntax

```
Set theStringCollection = theElement.GetDefaultSetNames (theToolName)
```

```
theStringCollection As StringCollection
```

Returns the names of the default property sets defined for the given element and tool name.

```
theElement As RoseRT.Element
```

Element whose default set names are being retrieved.

```
theToolName As String
```

Name of the tool whose default set names are being retrieved.

GetPropertyClassName () : String

Description

Retrieves the class name of a given element.

Syntax

```
theClassName = theElement.GetPropertyClassName ()
```

```
theClassName As String
```

Returns the class name for the given element.

```
theElement as RoseRT.Element
```

Element whose class name is being retrieved.

GetPropertyValue (theToolName : String, thePropName : String) : String

Description

Retrieves the current value of a property of an element, given a property and tool name.

Syntax

```
theValue = theElement.GetPropertyValue (theToolName, thePropName)
```

```
theValue As String
```

Returns the current value for the given tool and property .

```
theElement As RoseRT.Element
```

Element for which the property value is being retrieved.

```
theToolName As String
```

Name of the tool for which a property value is being retrieved.

```
thePropName As String
```

Name of the property whose value is being retrieved.

GetQualifiedName () : String

Description

Retrieves the qualified name of a model element.

The qualified name includes the names of the packages to which the element belongs. This allows the name to resolve to a specific class, since the Rose allows multiple classes of the same name to exist in a model, as long as they are in different packages.

Examples

- The qualified name of the ComponentPackageView Class is:
- Logical View::Physical Classes::ComponentPackageView
- The qualified name of the PathMap Class is: Logical View::Application Classes::PathMap

Syntax

```
Set theName = theElement.GetQualifiedName ()
```

```
theName As String
```

Returns the qualified name of the element.

```
theElement As RoseRT.Element
```

Element whose qualified name is being returned.

GetToolNames () : StringCollection

Description

Retrieves the names of the tools defined for the specified element.

Syntax

```
Set theStringCollection = theElement.GetToolNames
```

```
theStringCollection As StringCollection
```

Returns the names of the tools for the given element.

```
theElement As RoseRT.Element
```

Element whose tool names are being retrieved.

GetToolProperties (theToolName : String) : PropertyCollection

Description

Retrieves the properties for the given element and tool name.

Syntax

```
Set thePropertyCollection = theElement.GetToolProperties (theToolName)
```

```
thePropertyCollection As PropertyCollection
```

Returns the collection of properties defined for the specified tool name and element .

```
theElement As RoseRT.Element
```

Element whose tool properties are being retrieved.

GetUniqueID () : String

Description

Retrieves the unique ID for a model element. Each element in a model has a unique ID, which is set internally. You cannot set this value, but you can retrieve it.

Syntax

```
Set theUniqueID = theElement.GetUniqueID ()
```

`theUniqueId As String`

Returns the string value of the element's unique ID.

`theElement As RoseRT.Element`

Element whose ID is being returned.

InheritProperty (theToolName : String, thePropName : String) : Boolean

Description

Removes the overridden value from an element's property so that the default value is used . If there is no default value, then a call to the GetPropertyValue method on the inherited property returns an empty string.

Syntax

```
IsInherited = theElement.InheritProperty (theToolName, thePropName)
```

`IsInherited as Boolean`

Returns a value of True when the property is returned to its inherited (default) value.

`theElement As RoseRT.Element`

Element to which the property belongs.

`theToolName As String`

Name of the tool to which the property applies.

`thePropName As String`

Name of the property whose value is being inherited.

IsDefaultProperty (theToolName : String, thePropName : String) : Boolean

Description

Indicates whether the current value of a property is set to its default value.

Syntax

```
IsDefault = theElement.IsDefaultProperty (theToolName, thePropName)
```

```
IsDefault As Boolean
```

Returns a value of True if the current value of the property is set to its default value .

```
theElement As RoseRT.Element
```

The model element whose property value is being checked.

```
theToolName As String
```

Tool name to which the property applies.

```
thePropName As String
```

Name of the property whose default status is being checked.

IsOverriddenProperty (theToolName : String, thePropName : String) : Boolean

Description

Indicates whether the default value of a property is currently overridden by a different value.

Syntax

```
IsOverridden = theElement.IsOverriddenProperty (theToolName,
thePropName)
```

```
IsOverridden As Boolean
```

Returns a value of True if the default value of a property is currently overridden.

```
theElement As RoseRT.Element
```

The model element whose property value is being checked.

```
theToolName As String
```

Tool name to which the property applies.

```
thePropName As String
```

Name of the property whose overridden status is being checked.

OverrideProperty (theToolName : String, thePropName : String, theValue : String) : Boolean

Description

Overrides the default value of a element's property. If the given property does not exist in the default set, a new string type property is created for this element only.

Syntax

```
IsOverridden = theElement.OverrideProperty (theToolName, thePropName,
theValue)
IsOverrridden As Boolean
```

Returns a value of True when the property value is successfully overridden.

```
theElement as RoseRT.Element
```

Element to which the property applies.

```
theToolName As String
```

Name of the tool to which the property applies.

```
thePropName As String
```

Name of the property whose default value is being overridden.

```
theValue As String
```

Value being set in place of the default value.

SetCurrentPropertySetName (ToolName : String, SetName : String) : Boolean

Description

Specifies a given property set as the current property set for the element

Syntax

```
IsCurrentSet = theElement.SetCurrentPropertySetName (theToolName,
theSetName)
```

`IsCurrentSet As Boolean`

Returns a value of True when the given property set is set to the current property set for the element .

`theElement As RoseRT.Element`

Element whose current property set is being set.

`theToolname As String`

Name of the tool to which the property set applies.

`theSetName As String`

Name of the property set to become the current set.

ExternalDocument

Description

The ExternalDocument class exposes attributes and operations that allow you to create external documents (reports) from within the Rose RealTime environment. For example, you can start Word for Windows and output information from a Rose RealTime model into a Word document.

Derived from RRTEIObject

Public Attributes

ParentLogicalPackage : LogicalPackage

Description

Specifies the LogicalPackage that contains the external document.

Path : String

Description

Specifies the path to the external document.

Note: An external document is created with a type parameter of either Path or URL. When accessing an external document, you must specify the correct property (Path or URL) or a runtime error will occur. For example, you cannot access an external document whose type is Path by specifying a URL.

URL : String

Description

Specifies the Universal Resource Locator (URL) of an internet document.

Note: An external document is created with a type parameter of either Path or URL. When accessing an external document, you must specify the correct property (Path or URL), or a runtime error will occur. For example, you cannot access an external document whose type is URL by specifying a Path.

Public Operations

IsURL () : Boolean

Description

Checks whether the document is an internet document and therefore has a universal resource locator (URL).

Syntax

```
IsURL = theExternalDocument.IsURL ( )
```

```
IsURL As Boolean
```

Returns a value of true if the object has a URL.

```
theExternalDocument As RoseRT.ExternalDocument
```

Contains the document being checked.

Open (szAppPath : String) : Boolean

Description

Opens an external document based on a specified application path.

If you do not specify an application path, the Rose RealTime application attempts to locate and launch the application based on the external document's type (file extension).

For example, if the ExternalDocument is linked to a file with the .txt extension, and you have associated .txt files with the Notepad application, Rose RealTime attempts to locate and start Notepad and opens the .txt file that contains the external document.

Syntax

```
IsOpen = theExternalDocument.Open (AppPath)
```

```
IsOpen As Boolean
```

Returns a value of true when the specified document is successfully opened.

```
theExternalDocument As RoseRT.ExternalDocument
```

Document being opened.

```
AppPath As String
```

Path to the application executable being used to open the document.

Note: You can specify any appropriate application to open the document. For example, you can use Word or WordPad to open a .doc file.

Model

Description

Once you use the application class methods to set the current model, the model class provides attributes and operations that allow you to work with the objects in that model.

For example, you can:

- Add objects (classes, categories, relationships, processors, devices, diagrams, etc.) to the model
- Retrieve objects from the model
- Delete objects from the model

Check the lists of attributes and operations for complete information.

Note: In addition to the Model Class attributes and operations, all ModelElement operations that manipulate properties also apply to the Model Class.

Derived from Package

Public Attributes

ActiveComponent : Component

Description

Used to select an active component. When a component is configured as being active the toolbar build icons and menu items become available for easy access to common build and run commands.

DefaultProperties : DefaultModelProperties

Description

Collection of default properties belonging to the model.

DeploymentDiagram : DeploymentDiagram

Description

Specifies a deployment diagram belonging to the model.

RootComponentPackage : ComponentPackage

Description

ComponentPackage named <Top Level> in Rose RealTime. RootComponentPackage corresponds to the model's component view. This value can be retrieved, but not set.

RootDeploymentPackage : DeploymentPackage

RootLogicalPackage : LogicalPackage

Description

LogicalPackage named <Top Level> in Rose RealTime. RootLogicalPackage corresponds to the model's logical view. This value can be retrieved, but not set.

RootUseCaseLogicalPackage : LogicalPackage

Description

Root LogicalPackage to which the use cases belong. RootUseCaseLogicalPackage corresponds to the model's UseCase view. This value can be retrieved, but not set.

UseCases : UseCaseCollection

Description

Specifies the collection that contains the use cases that belong to the model

Public Operations

AddActiveComponentInstance (ComponentInstanceToAdd : ComponentInstance) : Boolean

Description

Adds a Component Instance to the collection of active Component Instances owned by the model. Notice the active component instance collection is actually stored in the Workspace.

Syntax

```
Added = theModel.AddActiveComponentInstance( ComponentInstanceToAdd )


Added As Boolean
```

Returns a value of True when the component instance has been successfully added to the active component instances collection.

```
theModel As RoseRT.Model
```

The model owning the active component instances collection from which the active component instance is being added to.

```
ComponentInstanceToAdd As RoseRT.ComponentInstance
```

The component instance to add to the active component instance collection.

AddDevice (pName : String) : Device

Description

Creates a new device and adds it to a model.

Syntax

```
Set theDevice = theModel.AddDevice (theName)
```

```
theDevice As RoseRT.Device
```

Returns the newly created device.

```
theModel As RoseRT.Model
```

Instance of the model to which the device is being added.

```
theName As String
```

Name of the device being added to the model.

AddProcessor (pName : String) : Processor

Description

Creates a new processor and adds it to a model.

Syntax

```
Set theProcessor = theModel.AddProcessor (theName)
```

```
theProcessor As RoseRT.Processor
```

Returns the processor being added to the model.

```
theModel As RoseRT.Model
```

Instance of the Processor being added to the model.

```
theName As String
```

Name of the Processor being added to the model.

ControlAllUnits (bControlAllUnits : Boolean) : Boolean

Description

Specifies whether the tool will load/save classes, packages and diagrams as individual files.

Syntax

```
UnitControlled = theModel.ControlAllUnits( bControlAllUnits )
```

```
UnitControlled As Boolean
```

Returns a value of True if the controlled units status was successfully set to bControllAllUnits.

```
theModel As RoseRT.Model
```

The model to set the controlled unit status.

```
bControlAllUnits As Boolean
```

The state to set the controlled unit status to.

DeleteDevice (pDevice : Device) : Boolean

Description

Deletes a device from a model.

Syntax

```
Deleted = theModel.DeleteDevice (theDevice)
```

```
Deleted As Boolean
```
Returns a value of True when the device is deleted.

```
theModel As RoseRT.Model
```
Instance of the model from which the device is being deleted.

```
theDevice As RoseRT.Device
```
Instance of the device being deleted.

DeleteProcessor (pProcessor : Processor) : Boolean

Description

Deletes a processor from a model.

Syntax

```
Deleted = theModel.DeleteProcessor (theProcessor)
```

```
Deleted As Boolean
```
Returns a value of True when the processor is deleted from the model.

```
theModel As RoseRT.Model
```
Instance of the model from which the processor is being deleted.

```
theProcessor As RoseRT.Processor
```
Instance of the processor being deleted.

FindCapsuleWithID (UniqueID : String) : Capsule

Description

Returns a specific capsule given the capsule's unique ID.

Syntax

```
Set theCapsule = theModel.FindCapsuleWithID (theUniqueID)
```

```
theCapsule As RoseRT.Capsule
```
Returns the capsule that corresponds to the given UniqueID.

```
theModel As RoseRT.Model
```
Model that contains the capsule.

```
theUniqueID As String
```
UniqueID of the capsule for which to search.

FindCapsules (CapsuleName : String) : CapsuleCollection

Description

Returns a collection of capsules belonging to the model.

Syntax

```
Set theCapsuleCollection = theModel.FindCapsules (CapsuleName)
```

```
theCapsuleCollection As RoseRT.CapsuleCollection
```
Returns a collection of capsules that match the given capsule name.

```
theModel As RoseRT.Model
```
Model that contains the capsules.

```
CapsuleName As String
```
Name of the capsule for which to search the model.

FindClassWithID (UniqueID : String) : Class

Description

Returns a specific class given the class's unique ID.

Syntax

```
Set theClass = theModel.FindClassWithID (theUniqueID)
```

```
theClass As RoseRT.Class
```

Returns the Class that corresponds to the given UniqueID.

```
theModel As RoseRT.Model
```

Model that contains the Class.

```
theUniqueID As String
```

UniqueID of the Class for which to search.

FindClasses (ClassName : String) : ClassCollection

Description

Returns a collection of classes belonging to the model.

Syntax

```
Set theClassCollection = theModel.FindClasses (theClassName)
```

```
theClassCollection As RoseRT.ClassCollection
```

Returns a collection of classes that match the given class name.

```
theModel As RoseRT.Model
```

Model that contains the classes.

```
theClassName As String
```

Name of the class for which to search the model.

FindLogicalPackageWithID (UniqueID : String) : LogicalPackage

Description

Returns a specific LogicalPackage given the LogicalPackage's unique ID.

Syntax

```
Set theLogicalPackage = theModel.FindLogicalPackageWithID
(theUniqueID)
```

```
theLogicalPackage As RoseRT.LogicalPackage
```

Returns the LogicalPackage that corresponds to the given UniqueID.

```
theModel As RoseRT.Model
```

Model that contains the LogicalPackage.

```
theUniqueID As String
```

UniqueID of the LogicalPackage for which to search.

FindLogicalPackages (LogicalPackageName : String) : LogicalPackageCollection

Description

Returns a collection of LogicalPackages belonging to the model.

Syntax

```
Set theLogicalPackageCollection = theModel.FindLogicalPackage
(theLogicalPackageName)
```

```
theLogicalPackageCollection As RoseRT.LogicalPackageCollection
```

Returns a collection of LogicalPackages that match the given LogicalPackage name.

```
theModel As RoseRT.Model
```

Model that contains the LogicalPackages.

```
theLogicalPackageName As String
```

Name of the LogicalPackage for which to search the model.

FindModelElementWithID (UniqueID : String) : ModelElement

Description

Returns a specific ModelElement given the ModelElement's unique ID.

Syntax

Set theModelElement = theModel.FindModelElementWithID (theUniqueID)

```
theModelElement As RoseRT.ModelElement
```

Returns the ModelElement that corresponds to the given UniqueID.

```
theModel As RoseRT.Model
```

Model that contains the ModelElement.

```
theUniqueID As String
```

UniqueID of the ModelElement for which to search.

FindModelElements (ModelElementName : String) : ModelElementCollection

Description

Returns a collection of ModelElements belonging to the model.

Syntax

```
Set theModelElementCollection = theModel.FindModelElements
(theModelElementName)
```

```
theModelElementCollection As RoseRT.ModelElementCollection
```

Returns a collection of ModelElements that match the given ModelElement name.

```
theModel As RoseRT.Model
```

Model that contains the ModelElements.

```
theModelElementName As String
```

Name of the ModelElement for which to search the model.

FindProtocolWithID (UniqueID : String) : Protocol

Description

Returns a specific protocol given the protocol's unique ID.

Syntax

```
Set theProtocol = theModel.FindProtocolWithID (theUniqueID)
```

`theProtocol As RoseRT.Protocol`

Returns the protocol that corresponds to the given UniqueID.

`theModel As RoseRT.Model`

Model that contains the protocol.

`theUniqueID As String`

UniqueID of the protocol for which to search.

FindProtocols (ProtocolName : String) : ProtocolCollection

Description

Returns a collection of protocols belonging to the model.

Syntax

```
Set theProtocolCollection = theModel.FindProtocols (ProtocolName)
```

`theProtocolCollection As RoseRT.ProtocolCollection`

Returns a collection of protocols that match the given protocol name.

`theModel As RoseRT.Model`

Model that contains the protocols.

`ProtocolName As String`

Name of the protocol for which to search the model.

GetActiveComponentInstances () : ComponentInstanceCollection

Description

Returns the collection of active Component Instances owned by a model. Notice the active component instance collection is actually stored in the Workspace.

Syntax

```
Set theActiveComponents = theModel.GetActiveComponentInstances()
```

```
theActiveComponents As RoseRT.ComponentInstanceCollection
```

Returns the collection of active Component Instances owned by the model.

```
theModel As RoseRT.Model
```

The model from which the active component instance collection is being retrieved from.

GetActiveDiagram () : Diagram

Description

Returns the currently active diagram from the current model. The active diagram is the window in Rose RealTime that currently has the focus.

Syntax

```
Set theDiagram = theModel.GetActiveDiagram ()
```

```
theDiagram As RoseRT.Diagram
```

Returns the currently active Rose RealTime diagram from the model. Returns nothing if a window that is not a diagram, such as a script window or the Browser, has the focus.

```
theModel As RoseRT.Model
```

Instance of the model from which the diagram is being retrieved.

GetAllAssociations () : AssociationCollection

Description

Returns all Associations belonging to all Logical Packages the model.

Syntax

```
Set theAssociations = theModel.GetAllAssociations()
```

```
theAssociations As RoseRT.AssociationCollection
```
The associations contained in theModel.

```
theModel As RoseRT.Model
```
Model to retrieve all the associations from.

GetAllCapsules () : CapsuleCollection

Description

Returns all Capsules belonging to all Logical Packages the model.

Syntax

```
Set theCapsules = theModel.GetAllCapsules()
```

```
theCapsules As RoseRT.CapsuleCollection
```
The capsules contained in theModel.

```
theModel As RoseRT.Model
```
Model to retrieve all the capsules from.

GetAllClasses () : ClassCollection

Description

Returns all classes belonging to all categories in the model.

Syntax

```
Set theClasses = theModel.GetAllClasses ()
```

```
theClasses As RoseRT.ClassCollection
```
Returns the collection of classes retrieved from the model.

```
theModel As RoseRT.Model
```
Instance of the model from which classes are being retrieved.

GetAllComponentPackages () : ComponentPackageCollection

Description

Returns all ComponentPackages belonging to the model.

Syntax

```
Set theComponentPackage = theModel.GetAllComponentPackage ()
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns the collection of ComponentPackage retrieved from the model.

```
theModel As RoseRT.Model
```

Instance of the model from which ComponentPackage are being retrieved.

GetAllComponents () : ComponentCollection

Description

Returns all components belonging to the model.

Syntax

```
Set theComponents = theModel.GetAllComponents ()
```

```
theComponents As RoseRT.ComponentCollection
```

Returns the collection of components retrieved from the model.

```
theModel As RoseRT.Model
```

Instance of the model from which components are being retrieved.

GetAllDevices () : DeviceCollection

Description

Returns all devices belonging to the model.

Syntax

```
Set theDevices = theModel.GetAllDevices ()
```

```
theDevices As RoseRT.DeviceCollection
```
Returns the collection of devices retrieved from the model.

```
theModel As RoseRT.Model
```
Instance of the model from which devices are being retrieved.

GetAllLogicalPackages () : LogicalPackageCollection

Description

Returns all LogicalPackages belonging to the model.

Syntax

```
Set theLogicalPackage = theModel.GetAllLogicalPackages ()
```

```
theLogicalPackagez As RoseRT.LogicalPackageCollection
```
Returns the collection of LogicalPackages retrieved from the model.

```
theModel As RoseRT.Model
```
Instance of the model from which LogicalPackages are being retrieved.

GetAllProcessors () : ProcessorCollection

Description

Returns all processors belonging to the model

Syntax

```
Set theProcessors = theModel.GetAllProcessors ()
```

```
theProcessors As RoseRT.ProcessorCollection
```
Returns the collection of processors retrieved from the model.

```
theModel As RoseRT.Model
```
Instance of the model from which processors are being retrieved.

GetAllProtocols () : ProtocolCollection

Description

Returns all Protocols belonging to all Logical Packages the model.

Syntax

```
Set theProtocols = theModel.GetAllProtocols()
```

```
theProtocols As RoseRT.ProtocolCollection
```

The protocols contained in theModel.

```
theModel As RoseRT.Model
```

Model to retrieve all the protocols from.

GetAllUseCases () : UseCaseCollection

Description

Returns all use cases belonging to the model.

Syntax

```
Set theUseCases = theModel.GetAllUseCases ()
```

```
theUseCases As RoseRT.UseCaseCollection
```

Returns the collection of use cases retrieved from the model.

```
theModel As RoseRT.Model
```

Instance of the model from which use cases are being retrieved.

GetSelectedCapsules () : CapsuleCollection

Description

Returns all capsules selected in the current model.

Syntax

```
Set theCapsules = theModel.GetSelectedCapsules ()
```

```
theCapsules As RoseRT.CapsuleCollection
```

Returns the collection of capsules currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which capsules are being retrieved.

GetSelectedClasses () : ClassCollection

Description

Returns all classes selected in the current model.

Syntax

```
Set theClasses = theModel.GetSelectedClasses ()
```

```
theClasses As RoseRT.ClassCollection
```

Returns the collection of classes currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which classes are being retrieved.

GetSelectedComponentPackages () : ComponentPackageCollection

Description

Returns all ComponentPackages selected in the current model.

Syntax

```
Set theComponentPackages = theModel.GetSelectedComponentPackages ()
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns the collection of ComponentPackages currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which ComponentPackages are being retrieved.

GetSelectedComponents () : ComponentCollection

Description

Returns all components selected in the current model.

Syntax

```
Set theComponents = theModel.GetSelectedComponents ()
```

```
theComponents As RoseRT.ComponentCollection
```

Contains the collection of components currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which components are being retrieved.

GetSelectedLogicalPackages () : LogicalPackageCollection

Description

Returns all LogicalPackages selected in the current model.

Syntax

```
Set theLogicalPackages = theModel.GetSelectedLogicalPackages ()
```

```
theLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns the collection of LogicalPackages currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which LogicalPackages are being retrieved.

GetSelectedModelElements () : ModelElementCollection

Description

Returns all model elements selected in the current model.

Syntax

```
Set theModelElements = theModel.GetSelectedModelElements()
```

```
theModelElements As RoseRT.ModelElementCollection
```

Returns the collection of model elements currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which model elements are being retrieved.

GetSelectedProtocols () : ProtocolCollection

Description

Returns all protocols selected in the current model.

Syntax

```
Set theProtocols = theModel.GetSelectedProtocols ()
```

```
theProtocols As RoseRT.ProtocolCollection
```

Returns the collection of protocols currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which protocols are being retrieved.

GetSelectedUseCases () : UseCaseCollection

Description

Returns all use cases selected in the current model.

Syntax

```
Set theUseCases = theModel.GetSelectedUseCases ()
```

```
theUseCases As RoseRT.UseCaseCollection
```

Returns the collection of use cases currently selected in the model.

```
theModel As RoseRT.Model
```

Instance of the model from which use cases are being retrieved.

RemoveActiveComponentInstance (ComponentInstanceToRemove : ComponentInstance) : Boolean

Description

Removes a Component Instance from the collection of active Component Instances owned by the model. Notice the active component instance collection is actually stored in the Workspace.

Syntax

```
Removed = theModel.RemoveActiveComponentInstance(
ComponentInstanceToRemove )
```

`Removed As Boolean`

Returns a value of True when the component instance has been successfully removed from the active component instances collection.

`theModel As RoseRT.Model`

The model owning the active component instances collection from which the active component instance is being removed from.

`ComponentInstanceToRemove As RoseRT.ComponentInstance`

The component instance to remove from the active component instance collection.

ModelElement

Description

Every ModelElement is a model element and therefore inherits all Element attributes and operations. Use ModelElement attributes and operations to specify or manipulate ModelElement documentation, stereotypes, external documents, as well as to open a ModelElement's specification

Derived from ControllableElement

Public Attributes

Documentation : String

Description

Specifies the documentation belonging to the ModelElement.

ExternalDocuments : ExternalDocumentCollection

Description

Specifies the external documents belonging to the ModelElement.

LocalizedStereotype : String

Description

Specifies the localized equivalent of the ModelElement stereotype.

Stereotype : String

Description

Specifies the stereotype of the ModelElement

Public Operations

AddExternalDocument (szName : String, iType : RsExternalDocumentType) : ExternalDocument

Description

Creates a new external document and adds it to a ModelElement.

Syntax

```
Added = theModelElement.AddExternalDocument (theName, theType)


Added As Boolean
```

Returns a value of true when the document is added to the ModelElement.

```
theModelElement As RoseRT.ModelElement
```

ModelElement to which the document is being added.

```
theName As String
```

Name of the document being added.

```
theType As Integer
```

Type of document being added Valid values are:

1 = Path

2 = URL

DeleteExternalDocument (pIDispatch : ExternalDocument) : Boolean

Description

Deletes an external document from a ModelElement.

Syntax

```
Deleted = theModelElement.DeleteExternalDocument (theDocument)
```

```
deleted As Boolean
```

Returns a value of true when the document is deleted from the ModelElement.

```
theModelElement As RoseRT.ModelElement
```

ModelElement from which the document is being deleted.

```
theDocument As RoseRT.ExternalDocument
```

Instance of the document being deleted.

GetModelElement () : ModelElement

Description

Retrieves a ModelElement as an object.

Note: Use this operation to convert classes derived from ModelElement into ModelElement objects.

Syntax

```
Set theModelElement = theObject.GetModelElement( )
```

```
theModelElement As RoseRT.ModelElement
```

Returns the Rose item as an object.

```
theModelElement As RoseRT.ModelElement
```

Instance of the ModelElement being returned.

OpenSpecification () : Boolean

Description

Opens the specification window for the specified ModelElement.

Syntax

```
Opened = theModelElement.OpenSpecification ( )
```

```
Opened As Boolean
```

Returns a value of TRUE when the specification is successfully opened.

```
theModelElement As RoseRT.ModelElement
```

ModelElement whose specification is being opened.

Package

Description

The Package Class is a container for the model elements that correspond to the UML Package concept.

Package class operations allow you to determine whether a package is the root package in a model, as well as to obtain the OLE object associated with the package.

Derived from ModelElement

Public Operations

AddSharedUnit (FileName : String) : Boolean

Description

Shares Model Elements from a unit in a Package.

Syntax

```
Added = thePackage.AddSharedUnit( FileName As String )
```

`Added As Boolean`

Returns True when successfully shared Model Elements of a unit into thePackage.

`thePackage As RoseRT.Package`

The package to share unit's Model Elements with.

`FileName As String`

The name of the shared unit file.

AddUnit (FileName : String) : Boolean

Description

Adds Model Elements from a unit in a Package.

Syntax

```
Added = thePackage.AddUnit( FileName As String )
```

`Added As Boolean`

Returns True when successfully added Model Elements of a unit into thePackage.

`thePackage As RoseRT.Package`

The package to add unit's Model Elements to.

```
FileName As String
```

The name of the unit file to add to the package.

ImportFile (FileName : String) : Boolean

Description

Imports Model Elements from a file and place them into a Package.

Syntax

```
Imported = thePackage.ImportFile( FileName As String )
```

```
Imported As Boolean
```

Returns True when successfully imported Model Elements into thePackage.

```
thePackage As RoseRT.Package
```

The package to put imported Model Elements into.

```
FileName As String
```

The name of the file to import.

ImportFileEx (FileName : String) : ControllableElementCollection

Description

Imports Model Elements from a file and place them into a Package.

Syntax

```
ImportedControllableElements = thePackage.ImportFile( FileName As
String )
```

```
ImportedControllableElements As RoseRT.ControllableElementCollection
```

Returns a collection containing the Controllable Elements imported into thePackage.

```
thePackage As RoseRT.Package
```

The package to put imported Model Elements into.

```
FileName As String
```

The name of the file to import.

IsRootPackage () : Boolean

Description

Finds out if the specified package is the root package (category) of the model.

Syntax

```
IsRoot = thePackage.IsRootPackage ()
```

```
IsRoot As  Boolean
```

Returns a value of True if the package is the root package (category) of the model.

```
thePackage As RoseRT.Package
```

Package being checked as root package.

TopLevel () : Boolean

Description

Returns whether the Package is the Root Package, i.e. direct child of the Model Package.

Syntax

```
IsTopLevel = thePackage.TopLevel()
```

```
IsTopLevel As Boolean
```

Returns a value of True when the package is a direct child of the Model Package

```
thePackage As RoseRT.Package
```

Package to determine whether it is the Top Level.

Property

Description

The Property class exposes a set of attributes and operations that

- Determine the characteristics of attributes in a model (for example, property name and type, as well as the development tool associated with the property).
- Allow you to retrieve attributes from a model.

Derived from RRTEIObject

Public Attributes

Name : String

Description

Indicates the name of the property (without specifying a path).

ToolName : String

Description

Corresponds to a tab in the property specification. A tool can be a programming language tool (such as C++), a user-defined add-in to Rational Rose RealTime, or some other tool.

Type : String

Description

Indicates the type of information stored by the property.

Values:

- String
- Integer
- Float
- Char
- Boolean

- Enumeration

Note: Other values may be valid if user-defined enumerated types exist.

Value : String

Description

Indicates the value of the property

RsExternalDocumentType

Description

Enumeration used in ModelElement::AddExternalDocument() to determine the location of the document added to the Model Element.

Public Attributes

rsFile : Integer = 1

Description

The document's location is specified using a file system specific path.

rsURL : Integer = 2

Description

The document's location is specified using a URL.

StructuredProperty

Description

This class allows easy parsing of Structured Properties. Structured properties are text properties with the following format:

[<section-name1> {section-default-value1}{section-default-value2}{...}

<field-name1>=<value1>

<field-name1>=<value2>

...]

[<section-name2 ...]

default-value

Derived from Property

Public Operations

GetFieldValue

Description

Returns the value stored in field of a section within the StructuredProperty. An empty string is returned if the field or section do not exist.

Syntax

```
FieldValue = theStructuredProperty.GetFieldValue ( SectionName,
FieldName )
```

`FieldValue As String`

Returns the value stored in field FieldName of section SectionName.

`theStructuredProperty As RoseRT.StructuredProperty`

The property to retrieve a field value from.

`SectionName As String`

The name of the section where a field named FieldName can be found. Passing an empty string is interpreted as a request to retrieve the string property value string that is not included in any section.

`FieldName As String`

The name of the field to retrieve a value from. Passing an empty string is interpreted as a request to retrieve the section's default value.

SetFieldValue

Description

Sets the value to store in a section's field within the StructuredProperty. The section and/or the field will get created if they do not exist within the structured property.

Syntax

```
theStructuredProperty.SetFieldValue ( SectionName, FieldName, Value )
```

```
theStructuredProperty As RoseRT.StructuredProperty
```

The property to set a section's field value.

```
SectionName As String
```

The name of the section where a field named FieldName can be found. Passing an empty string is interpreted as a request to set the string property value string that is not included in any section.

```
FieldName As String
```

The name of the field to set a value into. Passing an empty string is interpreted as a request to set the section's default value.

```
Value As String
```

The value to store in the section's field.

Deployment View Classes

Deployment View classes include

- *ComponentInstance* on page 249
    - Public Attributes

      AttachTo : Boolean on page 249

      Component : Component on page 249

      ConnectionDelay : Integer on page 250

      ConsolePort : Integer on page 250

LoadDelay : Integer on page 250

LoadOrder : Integer on page 250

LogsPort : Integer on page 250

MyProcessor : Processor on page 251

OperationMode : String on page 251

Priority : String on page 251

TargetTimeout : Integer on page 251

TOPort : Integer on page 251

UserParameters : String on page 252

- Public Operations

ConnectionDelay : Integer on page 250

Priority : String on page 251

GetDefaultOperationModes () : StringCollection on page 252

- *DeploymentPackage* on page 252
  - Public Attributes

DeploymentDiagrams : DeploymentDiagramCollection on page 253

DeploymentPackages : DeploymentPackageCollection on page 253

ParentDeploymentPackage : DeploymentPackage on page 253

  - Public Operations

AddDeploymentDiagram (name : String) : DeploymentDiagram on page 253

AddDeploymentPackage (theName : String) : DeploymentPackage on page 254

AddDevice (pName : String) : Device on page 254

AddProcessor (pName : String) : Processor on page 254

DeleteDeploymentDiagram (theDeploymentDiagram : DeploymentDiagram) : Boolean on page 254

DeleteDeploymentPackage (theDeploymentPackageToDelete : DeploymentPackage) : Boolean on page 255

DeleteDevice (pDevice : Device) : Boolean on page 256

DeleteProcessor (pProcessor : Processor) : Boolean on page 256

GetAllDevices () : DeviceCollection on page 256

GetAllProcessors () : ProcessorCollection on page 256

RelocateDeploymentDiagram (theDeploymentDiagram : DeploymentDiagram) : Boolean on page 256

RelocateDeploymentPackage (theDeploymentPackage : DeploymentPackage) : Boolean on page 256

RelocateDevice (theDevice : Device) : Boolean on page 257

RelocateProcessor (theProcessor : Processor) : Boolean on page 257

- *Device* on page 258
  - Public Attributes

    Characteristics : String on page 258

    ParentDeploymentPackage : DeploymentPackage on page 258
  - Public Operations

    AddDeviceConnection (theDevice : Device) : Boolean on page 259

    AddProcessorConnection (theProcessor : Processor) : Boolean on page 259

    GetConnectedDevices () : DeviceCollection on page 260

    GetConnectedProcessors () : ProcessorCollection on page 260

    RemoveDeviceConnection (theDevice : Device) : Boolean on page 260

    RemoveProcessorConnection (theProcessor : Processor) : Boolean on page 261
- *Processor* on page 262
  - Public Attributes

    Address : String on page 262

    CPU : String on page 262

    ComponentInstances : ComponentInstanceCollection on page 262

    OS : String on page 262

    ParentDeploymentPackage : DeploymentPackage on page 262

    ServerAddress : String on page 263

    UserScriptDirectory : String on page 263
  - Public Operations

AddComponentInstance (Name : String) : ComponentInstance on page 263

AddDeviceConnection (theDevice : Device) : Boolean on page 264

AddProcessorConnection (Processor : Processor) : Boolean on page 265

DeleteComponentInstance (theComponentInstance : ComponentInstance) : Boolean on page 265

GetConnectedDevices () : DeviceCollection on page 266

GetConnectedProcessors () : ProcessorCollection on page 266

RemoveDeviceConnection (theDevice : Device) : Boolean on page 266

RemoveProcessorConnection (theProcessor : Processor) : Boolean on page 267

ComponentInstance

Description

A component instance describes a runable instance of a component built on a particular processor.

Derived from ModelElement

Public Attributes

AttachTo : Boolean

Description

Determines whether the toolset is to automatically observe a Component Instance when it is loaded by the target control scripts.

Component : Component

Description

Component this Component Instance instantiates.

ConnectionDelay : Integer

Description:

An integer value representing the number of seconds to delay before attempting to connect to the target. This allows Purify time to instrument the executable as necessary. For a large module, you will need to adjust the connection delay to be more than the default of 60 seconds.

ConsolePort : Integer

Description

Specify a TCP/IP port number which can be used to connect to the Services Library command line debugger via a telnet window.

Note: Rose RealTime 6.0 restriction - the console port number must be the same as the Target observability port.

LoadDelay : Integer

Description

An integer value representing the number of X delay before the component instance is loaded or run.

LoadOrder : Integer

Description

An integer value representing the relative order in which this component instance will be loaded, or run, in relation to other component instances listed and selection in the Build Settings dialog.

LogsPort : Integer

Description

Specify a TCP/IP port number which can be used to connect to the log via a telnet window.

MyProcessor : Processor

OperationMode : String

Description

The Operation Mode specifies the target control configuration for the process. Options are:

Basic - Use the target control utilities to automatically load and run the component instance.

Debugger MSDEV - Use the target control utilities and load the executable using the Microsoft Visual Studio debugger.

Debugger Tornado - Use the target control utilities and load the executable using the Tornado debugger

Debugger xxgdb - Use the target control utilities and load the executable in the GNU xxgdb debugger (UNIX only).

Manual - the toolset will not attempt to load the executable. The user must manually load the executable.

Priority : String

Description:

Sets the priority the component instance will run at.

TargetTimeout : Integer

Description:

Number of seconds to wait for a response from the target before assuming something is wrong.

TOPort : Integer

Description

Specify a TCP/IP port number to use for connecting the toolset's execution environment to the target executable. The port number must not already be in use by another process.

UserParameters : String

Description

Represents command line arguments that are passed on the command line when the process is loaded.

Public Operations

GetDefaultOperationModes () : StringCollection

Description

Returns the default Operation Modes that can be used to set the OperationMode attribute.

Syntax

```
Set DefaultOperationModes =
theComponentInstance.GetDefaultOperationModes()
```

```
DefaultOperationModes As RoseRT.StringCollection
```

Returns an array of strings, each corresponding to a default Operation Mode.

```
theComponentInstance As RoseRT.ComponentInstance
```

The Component Instance to retrieve default Operation Modes for.

DeploymentPackage

Description

The deployment package allows you to define and manipulate collections of device, processors and deployment diagrams. They can even be nested.

Derived from Package

Public Attributes

DeploymentDiagrams : DeploymentDiagramCollection

Description

Deployment diagrams owned by the deployment package.

DeploymentPackages : DeploymentPackageCollection

Description

Deployment packages owned by the deployment package.

ParentDeploymentPackage : DeploymentPackage

Description

Deployment package owning the deployment package.

Public Operations

AddDeploymentDiagram (name : String) : DeploymentDiagram

Description

Adds a deployment diagram to the deployment package.

Syntax

```
Set theDeploymentDiagram = theDeploymentPackage.AddDeploymentDiagram(
name )
```

`theDeploymentDiagram As RoseRT.DeploymentDiagram`

Returns the new deployment diagram added to the deployment package.

`theDeploymentPackage As RoseRT.DeploymentPackage`

Deployment package to which a new deployment diagram is being added.

`name As String`

Name of the new deployment diagram added to the deployment package.

AddDeploymentPackage (theName : String) : DeploymentPackage

Description

Adds a deployment package to the deployment package.

Syntax

```
Set theNewDeploymentPackage =
theDeploymentPackage.AddDeploymentPackage( theName )
```

```
theNewDeploymentPackage As RoseRT.DeploymentPackage
```

Returns the new deployment package added to the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package to which a new deployment package is being added.

```
theName As String
```

Name of the new deployment package added to the deployment package.

AddDevice (pName : String) : Device

AddProcessor (pName : String) : Processor

DeleteDeploymentDiagram (theDeploymentDiagram : DeploymentDiagram) : Boolean

Description

Deletes a deployment diagram from the deployment package.

Syntax

```
Deleted = theDeploymentPackage.DeleteDeploymentDiagram(
theDeploymentDiagram )
```

```
Deleted As Boolean
```

Returns a value of True when the deployment diagram is successfully deleted from the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a deployment diagram is being deleted.

```
theDeploymentDiagram As RoseRT.DeploymentDiagram
```

Deployment diagram to delete from the deployment package.

DeleteDeploymentPackage (theDeploymentPackageToDelete : DeploymentPackage) : Boolean

Description

Deletes a deployment package from the deployment package.

Syntax

```
Deleted = theDeploymentPackage.DeleteDeploymentPackage(
theDeploymentPackageToDelete )
```

```
Deleted As Boolean
```

Returns a value of True when the deployment package is successfully deleted from the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a deployment package is being deleted.

```
theDeploymentPackageToDelete As RoseRT.DeploymentPackage
```

Deployment package to delete from the deployment package.

DeleteDevice (pDevice : Device) : Boolean

DeleteProcessor (pProcessor : Processor) : Boolean

GetAllDevices () : DeviceCollection

GetAllProcessors () : ProcessorCollection

RelocateDeploymentDiagram (theDeploymentDiagram : DeploymentDiagram) : Boolean

Description

Relocates a deployment diagram into the deployment package.

Syntax

```
Relocated = theDeploymentPackage.RelocateDeploymentDiagram(
theDeploymentDiagram )
```

```
Relocated As Boolean
```

Returns a value of True when the deployment diagram is successfully relocated into the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a deployment diagram is being relocated into.

```
theDeploymentDiagram As RoseRT.DeploymentDiagram
```

Deployment diagram to relocate into the deployment package.

RelocateDeploymentPackage (theDeploymentPackage : DeploymentPackage) : Boolean

Description

Relocates a deployment package into the deployment package.

Syntax

```
Relocated = theDeploymentPackage.RelocateDeploymentPackage(
theRelocatedDeploymentPackage )
```

```
Relocated As Boolean
```

Returns a value of True when the deployment package is successfully relocated into the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a deployment package is being relocated into.

```
theRelocatedDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package to relocate into the deployment package.

RelocateDevice (theDevice : Device) : Boolean

Description

Relocates a device into the deployment package.

Syntax

```
Relocated = theDeploymentPackage.RelocateDevice( theDevice )
```

```
Relocated As Boolean
```

Returns a value of True when the device is successfully relocated into the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a device is being relocated into.

```
theDevice As RoseRT.Device
```

Device to relocate into the deployment package.

RelocateProcessor (theProcessor : Processor) : Boolean

Description

Relocates a processor into the deployment package.

Syntax

```
Relocated = theDeploymentPackage.RelocateProcessor( theProcessor )
```

```
Relocated As Boolean
```

Returns a value of True when the processor is successfully relocated into the deployment package.

```
theDeploymentPackage As RoseRT.DeploymentPackage
```

Deployment package from which a processor is being relocated into.

```
theProcessor As RoseRT.Processor
```

Processor to relocate into the deployment package.

Device

Description

A device is hardware that is not capable of executing a program (a printer, for example). The device class exposes properties and methods that allow you to define and manipulate the characteristics of devices. Check the lists of attributes and operations for complete information.

Derived from ModelElement

Public Attributes

Characteristics : String

Description

Specifies the characteristics of the device

ParentDeploymentPackage : DeploymentPackage

Description

Deployment Package that owns this device.

Public Operations

AddDeviceConnection (theDevice : Device) : Boolean

Description

Creates a new device connection and adds it to the device.

Syntax

```
Connected = theDevice.AddDeviceConnection (theDevice)
```

`Connected As Boolean`

Returns a value of True when the device is connected.

`theDevice As RoseRT.Device`

Device to which the connection is being added.

`theDevice As RoseRT.Device`

Device at the other end of the connection being added.

AddProcessorConnection (theProcessor : Processor) : Boolean

Description

Creates a new device processor and adds it to the device.

Syntax

```
Connected = theDevice.AddProcessorConnection (theProcessor)
```

`Connected As Boolean`

Returns a value of True when the processor is connected.

`theDevice As RoseRT.Device`

Device to which the connection is being added.

`theProcessor As RoseRT.Processor`

Processor at the other end of the connection being added.

GetConnectedDevices () : DeviceCollection

Description

Retrieves the collection of devices that are connected to the device.

Syntax

```
Set theDevices = theDevice.GetConnectedDevices ( )
```

`theDevices As RoseRT.DeviceCollection`

Returns the collection of devices belonging to the device.

`theDevice As RoseRT.Device`

Device whose connected devices are being retrieved.

GetConnectedProcessors () : ProcessorCollection

Description

Retrieves the collection of processors that are connected to this device.

Syntax

```
Set theProcessors = theDevice.GetConnectedProcessors ( )
```

`theProcessors As RoseRT.ProcessorCollection`

Returns the collection of processors that are connected to the specified processor.

`theDevice As RoseRT.Device`

Device whose connected processors are being retrieved.

RemoveDeviceConnection (theDevice : Device) : Boolean

Description

Removes a device connection from the device.

Syntax

```
Removed = theDevice.RemoveDeviceConnection (theDevice)
```

`Removed As Boolean`

Returns a value of True when the device connection is removed.

`theDevice As RoseRT.Device`

Device from which the connection is being removed.

`theDevice As RoseRT.Device`

Device connection being removed.

RemoveProcessorConnection (theProcessor : Processor) : Boolean

Description

Removes a processor connection from the device.

Syntax

```
Removed = theDevice.RemoveProcessorConnection (theProcessor)
```

`Removed As Boolean`

Returns a value of True when the processor connection is removed.

`theDevice As RoseRT.Device`

Device from which the connection is being removed.

`theProcessor As RoseRT.Processor`

Processor connection being removed.

Processor

Description

A processor is hardware that is capable of executing programs. Processors are assigned to implement Component Instances.

Derived from ModelElement

Public Attributes

Address : String

Description

Network address for the processor, this field can contain a hostname, or an IP address. For example jhostl or 145.34.5.6.

Note: For systems not connected to a network, you must use 127.0.0.1 in this field.

CPU : String

Description

Name of the type of central processing unit for this processor element.

ComponentInstances : ComponentInstanceCollection

Description

List of component instances that will run on this processor

OS : String

Description

Name of the operating system running on this processor.

ParentDeploymentPackage : DeploymentPackage

Description

Deployment Package that owns this processor.

ServerAddress : String

Description

In some environments there is a server that handles loading, executing of a component instance for the target RTOS. This is the name or the address of this server.

UserScriptDirectory : String

Description

Path to the target control utility directory which contains the scripts and programs that are responsible for loading and unloading processes on that processor. If this property does not point to a valid script directory you won't be able to execute component instances from within the toolset.

Public Operations

AddComponentInstance (Name : String) : ComponentInstance

Description

Creates a new Component Instance to ran on a Processor. Notice that you should associate a Component with the Component Instance by setting the Component Instance's Component Property immediately after this creation. Undetermined behavior may occur otherwise.

Syntax

```
Set theComponentInstance = theProcessor.AddComponentInstance( Name )
```

```
theComponentInstance As RoseRT.ComponentInstance
```

Returns a new Component Instance to ran on theProcessor. The Component Instance is not associated with any Component at this point and should not be used until such an association is created by assigning a Component to the Component Instance's Component attribute.

```
theProcessor As RoseRT.Processor
```

The Processor to add a new Component Instance to.

```
Name As String
```

The new Component Instance's Name.

Example

```
Dim co As RoseRT.Component
Set co =
RoseRTApp.CurrentModel.RootComponentPackage.Components.GetAt(1)

Dim pr As RoseRT.Processor
Set pr = RoseRTApp.CurrentModel.GetAllProcessors().GetAt(1)

Dim ci As RoseRT.ComponentInstance
Set ci = pr.AddComponentInstance( "MyComponentInstance" )

Set ci.Component = co
```

AddDeviceConnection (theDevice : Device) : Boolean

Description

Creates a new device connection and adds it to the processor.

Syntax

```
DeviceConnectionAdded = theProcessor.AddDeviceConnection( theDevice )
```

`DeviceConnectionAdded As Boolean`

Returns a value of True when the device is connected

`theProcessor As RoseRT.Processor`

The Processor to which the connection is being added

`theDevice As RoseRT.Device`

Device to add connection to.

AddProcessorConnection (Processor : Processor) : Boolean

Description

Creates a new processor connection and adds it to the processor.

Syntax

```
ProcessorConnectionAdded = theProcessor.AddProcessorConnection(
Processor )
```

`ProcessorConnectionAdded As Boolean`

Returns a value of True when the processor is connected

`theProcessor As RoseRT.Processor`

The Processor to which the connection is being added

`Processor As RoseRT.Processor`

Processor to add connection to.

DeleteComponentInstance (theComponentInstance : ComponentInstance) : Boolean

Description

Deletes a Component Instance from a processor.

Syntax

```
ComponentInstanceDeleted = theProcessor.DeleteComponentInstance(
theComponentInstance )
```

`ComponentInstanceDeleted As Boolean`

Returns a value of True when the Component Instance is deleted

`theProcessor As RoseRT.Processor`

The Processor from which the Component Instance is being deleted

`theComponentInstance As RoseRT.ComponentInstance`

The Component Instance to delete from theProcessor.

GetConnectedDevices () : DeviceCollection

Description

Retrieves the collection of devices that are connected to this processor.

Syntax

```
Devices = theProcessor.GetConnectedDevices()
```

```
Devices As RoseRT.DeviceCollection
```

Returns the collection of devices that are connected to theProcessor.

```
theProcessor As RoseRT.Processor
```

The Processor whose connected devices are being retrieved.

GetConnectedProcessors () : ProcessorCollection

Description

Retrieves the collection of processors that are connected to this processor.

Syntax

```
Processors = theProcessor.GetConnectedProcessors()
```

```
Processors As RoseRT.ProcessorCollection
```

Returns the collection of processors that are connected to theProcessor.

```
theProcessor As RoseRT.Processor
```

The Processor whose connected processors are being retrieved.

RemoveDeviceConnection (theDevice : Device) : Boolean

Description

Removes a device connection from a processor.

Syntax

```
Removed = theProcessor.RemoveDeviceConnection( theDevice )
```

```
Removed As Boolean
```

Returns a value of True when the device connection is removed.

```
theProcessor As RoseRT.Processor
```

The Processor from which the connection is being removed.

```
theDevice As RoseRT.Device
```

The device to remove a connection to.

RemoveProcessorConnection (theProcessor : Processor) : Boolean

Description

Removes a processor connection from a processor.

Syntax

```
Removed = theProcessor.RemoveProcessorConnection( theProcessor )
```

```
Removed As Boolean
```

Returns a value of True when the processor connection is removed.

```
theProcessor As RoseRT.Processor
```

The Processor from which the connection is being removed.

```
theProcessor As RoseRT.Processor
```

The processor to remove a connection to.

Logical View Classes

Logical View classes include

- *LogicalPackage* on page 269

- Public Attributes

  Associations : AssociationCollection on page 270

  Capsules : CapsuleCollection on page 270

  ClassDiagrams : ClassDiagramCollection on page 270

  Classes : ClassCollection on page 270

  Collaborations : CollaborationCollection on page 270

  Global : Boolean on page 270

  LogicalPackages : LogicalPackageCollection on page 270

  ParentLogicalPackage : LogicalPackage on page 271

  Protocols : ProtocolCollection on page 271

  UseCases : UseCaseCollection on page 271

- Public Operations

  AddCapsule (name : String) : Capsule on page 271

  AddClass (theName : String) : Class on page 272

  AddClassDiagram (name : String) : ClassDiagram on page 272

  AddCollaboration (name : String) : Collaboration on page 273

  AddGeneralization (theRelationName : String, theParentLogicalPackageName : String) : Generalization on page 273

  AddLogicalPackage (theName : String) : LogicalPackage on page 274

  AddLogicalPackageDependency (theName : String, theSupplierLogicalPackageName : String) : LogicalPackageDependency on page 274

  AddProtocol (name : String) : Protocol on page 275

  AddUseCase (szName : String) : UseCase on page 275

  DeleteCapsule (theCapsule : Capsule) : Boolean on page 276

  DeleteClass (theClass : Class) : Boolean on page 277

  DeleteClassDiagram (theClassDiagram : ClassDiagram) : Boolean on page 277

  DeleteCollaboration (theCollaboration : Collaboration) : Boolean on page 278

  DeleteGeneralization (theGeneralization : Generalization) : Boolean on page 278

DeleteLogicalPackage (theLogicalPackage : LogicalPackage) : Boolean on page 279

DeleteLogicalPackageDependency (theDependency : LogicalPackageDependency) : Boolean on page 279

DeleteProtocol (theProtocol : Protocol) : Boolean on page 280

DeleteUseCase (theUseCase : UseCase) : Boolean on page 280

GetAllCapsules () : CapsuleCollection on page 281

GetAllClasses () : ClassCollection on page 281

GetAllLogicalPackages () : LogicalPackageCollection on page 282

GetAllProtocols () : ProtocolCollection on page 282

GetAllUseCases () : UseCaseCollection on page 282

GetAssignedComponentPackage () : ComponentPackage on page 283

GetGeneralizations () : GeneralizationCollection on page 283

GetLogicalPackageDependencies () : LogicalPackageDependencyCollection on page 283

GetSubLogicalPackages () : LogicalPackageCollection on page 284

GetSuperLogicalPackages () : LogicalPackageCollection on page 284

HasAssignedComponentPackage () : Boolean on page 285

RelocateCapsule (theCapsule : Capsule) : Boolean on page 285

RelocateClass (theClass : Class) : on page 285

RelocateClassDiagram (theClsDiagram : ClassDiagram) : on page 286

RelocateCollaboration (theCollaboration : Collaboration) : Boolean on page 286

RelocateLogicalPackage (theLogicalPackage : LogicalPackage) : on page 287

RelocateProtocol (theProtocol : Protocol) : Boolean on page 288

SetAssignedComponentPackage (newValue : ComponentPackage) : on page 288

LogicalPackage

Description

The logical package allows you to define and manipulate logical collections of classifiers, collaborations and diagrams.

Derived from Package

Public Attributes

Associations : AssociationCollection

Description

Associations owned by the logical package.

Capsules : CapsuleCollection

Description

Capsules owned by the logical package.

ClassDiagrams : ClassDiagramCollection

Description

Class diagrams owned by the logical package.

Classes : ClassCollection

Description

Classes owned by the logical package.

Collaborations : CollaborationCollection

Description

Collaborations owned by the logical package.

Global : Boolean

Description

Indicates that all public classes in the logical package can be used by any other logical package.

LogicalPackages : LogicalPackageCollection

Description

Logical packages owned by the logical package.

ParentLogicalPackage : LogicalPackage

Description

Logical package owning the logical package.

Protocols : ProtocolCollection

Description

Protocols owned by the logical package.

UseCases : UseCaseCollection

Description

Use cases owned by the logical package.

Public Operations

AddCapsule (name : String) : Capsule

Description

Adds a capsule to the logical package.

Syntax

```
Set theCapsule = theLogicalPackage.AddCapsule( name )
```

`theCapsule As RoseRT.Capsule`

Returns the new capsule added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package to which a new capsule is being added.

`name As String`

Name of the new capsule added to the logical package.

AddClass (theName : String) : Class

Description

Adds a class to the logical package.

Syntax

```
Set theClass = theLogicalPackage.AddClass( theName )
```

`theClass As RoseRT.Class`

Returns the new class added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package to which a new class is being added.

`theName As String`

Name of the new class added to the logical package.

AddClassDiagram (name : String) : ClassDiagram

Description

Adds a class diagram to the logical package.

Syntax

```
Set theClassDiagram = theLogicalPackage.AddClassDiagram( name )
```

`theClassDiagram As RoseRT.ClassDiagram`

Returns the new class diagram added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package to which a new class diagram is being added.

`name As String`

Name of the new class diagram added to the logical package.

AddCollaboration (name : String) : Collaboration

Description

Adds a collaboration to the logical package.

Syntax

```
Set theCollaboration = theLogicalPackage.AddCollaboration( name )
```

`theCollaboration As RoseRT.Collaboration`

Returns the new collaboration added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package to which a new collaboration is being added.

`name As String`

Name of the new collaboration added to the logical package.

AddGeneralization (theRelationName : String, theParentLogicalPackageName : String) : Generalization

Description

Adds a Generalization relationship to a Logical Package and returns it in the specified object.

Syntax

```
Set theGeneralization = theLogicalPackage.AddGeneralization(
theRelationName, theParentLogicalPackageName )
```

`theGeneralization As RoseRT.Generalization`

Returns the Generalization being added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical Package to which the Generalization is being added.

```
theRelationName As String
```

Name of the new Generalization.

```
theParentLogicalPackageName As String
```

Name of the parent logical package in the Generalize relationship.

AddLogicalPackage (theName : String) : LogicalPackage

Description

Adds a logical package to the logical package.

Syntax

```
Set theLogicalPackage = theLogicalPackage.AddLogicalPackage( theName )
```

```
theLogicalPackage As RoseRT.LogicalPackage
```

Returns the new logical package added to the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package to which a new logical package is being added.

```
theName As String
```

Name of the new logical package added to the logical package.

AddLogicalPackageDependency (theName : String, theSupplierLogicalPackageName : String) : LogicalPackageDependency

Description

Adds a logical package dependency relation to the logical package.

Syntax

```
Set theLogicalPackageDependency =
theLogicalPackage.AddLogicalPackageDependency( theName,
theSupplierLogicalPackageName )
```

```
theLogicalPackageDependency As RoseRT.LogicalPackageDependency
```

Returns the new logical package dependency added to the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package to which a new logical package dependency is being added.

```
theName As String
```

Name of the new logical package dependency added to the logical package.

```
theSupplierLogicalPackageName As String
```

Name of the logical package that theLogicalPackage is client of.

AddProtocol (name : String) : Protocol

Description

Adds a protocol to the logical package.

Syntax

```
Set theProtocol = theLogicalPackage.AddProtocol( name )
```

```
theProtocol As RoseRT.Capsule
```

Returns the new protocol added to the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package to which a new protocol is being added.

```
name As String
```

Name of the new protocol added to the logical package.

AddUseCase (szName : String) : UseCase

Description

Adds a use case to the logical package.

Syntax

```
Set theUseCase = theLogicalPackage.AddUseCase( szName )
```

`theUseCase As RoseRT.UseCase`

Returns the new use case added to the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package to which a new use case is being added.

`szName As String`

Name of the new use case added to the logical package.

DeleteCapsule (theCapsule : Capsule) : Boolean

Description

Deletes a capsule from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteCapsule( theCapsule )
```

`Deleted As Boolean`

Returns a value of True when the capsule is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a capsule is being deleted.

`theCapsule As RoseRT.Capsule`

Capsule to delete from the logical package.

DeleteClass (theClass : Class) : Boolean

Description

Deletes a class from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteClass( theClass )
```

`Deleted As Boolean`

Returns a value of True when the class is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a class is being deleted.

`theClass As RoseRT.Class`

Class to delete from the logical package.

DeleteClassDiagram (theClassDiagram : ClassDiagram) : Boolean

Description

Deletes a class diagram from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteClassDiagram( theClass )
```

`Deleted As Boolean`

Returns a value of True when the class diagram is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a class diagram is being deleted.

`theClassDiagram As RoseRT.ClassDiagram`

Class diagram to delete from the logical package.

DeleteCollaboration (theCollaboration : Collaboration) : Boolean

Description

Deletes a collaboration from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteCollaboration( theCollaborations )
```

`Deleted As Boolean`

Returns a value of True when the collaboration is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a collaboration is being deleted.

`theCollaboration As RoseRT.Collaboration`

Collaboration to delete from the logical package.

DeleteGeneralization (theGeneralization : Generalization) : Boolean

Description

Deletes a Generalization relation from a logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteGeneralization( theGeneralization )
```

`Deleted As Boolean`

Returns a value of True when the generalization gets deleted successfully from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical Package from which the generalization is being deleted.

`theGeneralization As RoseRT.Generalization`

The generalization being deleted.

DeleteLogicalPackage (theLogicalPackage : LogicalPackage) : Boolean

Description

Deletes a logical package from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteLogicalPackage(
theLogicalPackageToDelete )
```

`Deleted As Boolean`

Returns a value of True when the logical package is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a logical package is being deleted.

`theLogicalPackageToDelete As RoseRT.LogicalPackage`

Logical package to delete from the logical package.

DeleteLogicalPackageDependency (theDependency : LogicalPackageDependency) : Boolean

Description

Deletes a logical package dependency from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteLogicalPackageDependency(
theDependency )
```

`Deleted As Boolean`

Returns a value of True when the logical package dependency is successfully deleted from the logical package.

`theLogicalPackage As RoseRT.LogicalPackage`

Logical package from which a logical package dependency is being deleted.

```
theDependency As RoseRT.LogicalPackageDependency
```

Logical package dependency to delete from the logical package.

DeleteProtocol (theProtocol : Protocol) : Boolean

Description

Deletes a protocol from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteProtocol( theProtocol )
```

```
Deleted As Boolean
```

Returns a value of True when the protocol is successfully deleted from the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a protocol is being deleted.

```
theProtocol As RoseRT.Protocol
```

Protocol to delete from the logical package.

DeleteUseCase (theUseCase : UseCase) : Boolean

Description

Deletes a use case from the logical package.

Syntax

```
Deleted = theLogicalPackage.DeleteUseCase( theUseCase )
```

```
Deleted As Boolean
```

Returns a value of True when the use case is successfully deleted from the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a use case is being deleted.

```
theUseCase As RoseRT.Protocol
```

Use case to delete from the logical package.

GetAllCapsules () : CapsuleCollection

Description

Returns all capsules owned by the logical package and any of its subpackages.

Syntax

```
Set theCapsules = theLogicalPackage.GetAllCapsules()
```

```
theCapsules As RoseRT.CapsuleCollection
```

Returns a collection containing all capsules owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which capsules are being retrieved from.

GetAllClasses () : ClassCollection

Description

Returns all classes owned by the logical package and any of its subpackages.

Syntax

```
Set theClasses = theLogicalPackage.GetAllClasses()
```

```
theClasses As RoseRT.ClassCollection
```

Returns a collection containing all classes owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which classes are being retrieved from.

GetAllLogicalPackages () : LogicalPackageCollection

Description

Returns all logical packages owned by the logical package and any of its subpackages.

Syntax

```
Set theLogicalPackages = theLogicalPackage.GetAllLogicalPackages()
```

```
theLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns a collection containing all logical packages owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which logical packages are being retrieved from.

GetAllProtocols () : ProtocolCollection

Description

Returns all protocols owned by the logical package and any of its subpackages.

Syntax

```
Set theProtocols = theLogicalPackage.GetAllProtocols()
```

```
theProtocols As RoseRT.ProtocolCollection
```

Returns a collection containing all protocols owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which protocols are being retrieved from.

GetAllUseCases () : UseCaseCollection

Description

Returns all use cases owned by the logical package and any of its subpackages.

Syntax

```
Set theUseCases = theLogicalPackage.GetAllUseCases()
```

```
theUseCases As RoseRT.UseCaseCollection
```

Returns a collection containing all use cases owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which use cases are being retrieved from.

GetAssignedComponentPackage () : ComponentPackage

Description

Do not use, obsolete.

GetGeneralizations () : GeneralizationCollection

Description

Returns the set of Generalization a Logical Package is client of.

Syntax

```
Set Generalizations = theLogicalPackage.GetGeneralizations()
```

```
Generalizations As RoseRT.GeneralizationCollection
```

The collection of all Generalization relationships the Logical Package is client of.

```
theLogicalPackage As RoseRT.LogicalPackage
```

The Logical Package to return Generalization it is client of.

GetLogicalPackageDependencies () : LogicalPackageDependencyCollection

Description

Returns all logical package dependencies owned by the logical package and any of its subpackages.

Syntax

```
Set theLogicalPackageDependencies =
theLogicalPackage.GetAllLogicalPackagesDependencies()
```

```
theLogicalPackageDependencies As
RoseRT.LogicalPackageDependencyCollection
```

Returns a collection containing all logical packages dependencies owned by the logical package and any of its subpackages.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which logical packages dependencies are being retrieved from.

GetSubLogicalPackages () : LogicalPackageCollection

Description

Retrieves the sub logical packages derived from the logical package.

Syntax

```
Set theSubLogicalPackages = theLogicalPackage.GetSubLogicalPackages (
)
```

```
theSubLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns the collection of sub logical packages derived from the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical Package from which the collection is being retrieved.

GetSuperLogicalPackages () : LogicalPackageCollection

Description

Retrieves the super logical packages parent of the logical package.

Syntax

```
Set theSuperLogicalPackages =
theLogicalPackage.GetSuperLogicalPackages ( )
```

```
theSuperLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns the collection of super logical packages parent of the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical Package from which the collection is being retrieved.

HasAssignedComponentPackage () : Boolean

Description

Do not use, obsolete.

RelocateCapsule (theCapsule : Capsule) : Boolean

Description

Relocates a capsule into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateCapsule( theCapsule )
```

```
Relocated As Boolean
```

Returns a value of True when the capsule is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a capsule is being relocated into.

```
theCapsule As RoseRT.Capsule
```

Capsule to relocate into the logical package.

RelocateClass (theClass : Class) :

Description

Relocates a class into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateClass( theClass )
```

```
Relocated As Boolean
```

Returns a value of True when the class is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a class is being relocated into.

```
theClass As RoseRT.Class
```

Class to relocate into the logical package.

RelocateClassDiagram (theClsDiagram : ClassDiagram) :

Description

Relocates a class diagram into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateClassDiagram( theClsDiagram )
```

```
Relocated As Boolean
```

Returns a value of True when the class diagram is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a class diagram is being relocated into.

```
theClsDiagram As RoseRT.ClassDiagram
```

Class diagram to relocate into the logical package.

RelocateCollaboration (theCollaboration : Collaboration) : Boolean

Description

Relocates a collaboration into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateCollaboration( thecollaboration
)
```

```
Relocated As Boolean
```

Returns a value of True when the collaboration is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a collaboration is being relocated into.

```
theCollaboration As RoseRT.Collaboration
```

Collaboration to relocate into the logical package.

RelocateLogicalPackage (theLogicalPackage : LogicalPackage) :

Description

Relocates a logical package into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateLogicalPackage(
theLogicalPackage  )
```

```
Relocated As Boolean
```

Returns a value of True when the logical package is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a logical package is being relocated into.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package to relocate into the logical package.

RelocateProtocol (theProtocol : Protocol) : Boolean

Description

Relocates a protocol into the logical package.

Syntax

```
Relocated = theLogicalPackage.RelocateClass( theProtocol )
```

```
Relocated As Boolean
```

Returns a value of True when the class is successfully relocated into the logical package.

```
theLogicalPackage As RoseRT.LogicalPackage
```

Logical package from which a class is being relocated into.

```
theClass As RoseRT.Class
```

Class to relocate into the logical package.

SetAssignedComponentPackage (newValue : ComponentPackage) :

Description

Do not use, obsolete.

Association Classes

Association Classes include

- *Association* on page 290
    - Public Attributes

      AssociationClass : Class on page 290

      Derived : Boolean on page 290

      End1 : AssociationEnd on page 290

      End2 : AssociationEnd on page 291

      Ends : AssociationEndCollection on page 291

- Public Operations

  ClearAssociationEndForNameDirection () : on page 291

  GetAssociationEndForNameDirection () : AssociationEnd on page 291

  GetCorrespondingAssociationEnd (Classifier : Classifier) : AssociationEnd on page 292

  GetOtherAssociationEnd (Classifier : Classifier) : AssociationEnd on page 292

  NameIsDirectional () : Boolean on page 293

  SetAssociationEndForNameDirection (theAssociationEnd : AssociationEnd) : on page 293

  SetEnds (End1 : ModelElement, End2 : ModelElement) : Boolean on page 293

- *AssociationEnd* on page 294
  - Public Attributes

    Aggregate : Boolean on page 294

    AssociateModelElement : ModelElement on page 295

    Association : Association on page 295

    Classifier : Classifier on page 295

    Constraints : String on page 295

    Containment : AssociationEndContainment on page 295

    Friend : Boolean on page 295

    Keys : AttributeCollection on page 295

    Multiplicity : String on page 296

    Navigable : Boolean on page 296

    Static : Boolean on page 296

    UseCase : UseCase on page 296

    Visibility : AssociationEndVisibilityKind

  - Public Operations

    AddKey (theName : String, theType : String) : Attribute on page 296

    DeleteKey (theAttr : Attribute) : Boolean on page 297

    GetClassName () : String on page 297

IsAssociateClass () : Boolean on page 298

- *AssociationEndContainment* on page 298
- *AssociationEndVisibilityKind* on page 299

Association

Description

An association is a connection, or a link, between classes. The association class exposes a set of attributes and operations that

- Determine the characteristics of associations between classes
- Allow you to retrieve associations from a model

Check the lists of attributes and operations for complete information.

Derived from ModelElement

Public Attributes

AssociationClass : Class

Description

Class holding attributes and operations of an Association Class. May point to nothing if the Association is not an Association Class.

Derived : Boolean

Description

Indicates whether this object is derived from another object.

End1 : AssociationEnd

Description

Specifies an object as being End1 in an association.

End2 : AssociationEnd

Description

Specifies an object as being End2 in an association.

Ends : AssociationEndCollection

Description

Specifies the collection of AssociationEnds belonging to the Association.

Public Operations

ClearAssociationEndForNameDirection () :

Description

Clears name direction setting for the association.

Syntax

```
theAssociation.ClearAssociationEndForNameDirectionn


theAssociation As RoseRT.Association
```

The association to clear the association end.

GetAssociationEndForNameDirection () : AssociationEnd

Description

Retrieves the AssociationEnd that is set as the name direction for the association.

Syntax

```
Set theAssociationEnd =
theAssociation.GetAssociationEndForNameDirection ( )


theAssociationEnd As RoseRT.AssociationEnd
```

Returns the AssociationEnd that is set as the association's name direction.

```
theAssociation As RoseRT.Association
```

Association from which the AssociationEnd is being retrieved.

GetCorrespondingAssociationEnd (Classifier : Classifier) : AssociationEnd

Description

Retrieves the AssociationEnd associated with a specified class.

Syntax

```
Set theAssociationEnd = theAssociation.GetCorrespondingAssociationEnd
(theClass)
```

```
theAssociationEnd As RoseRT.AssociationEnd
```

Returns the AssociationEnd that corresponds to the specified class.

```
theAssociationEnd As RoseRT.AssociationEnd
```

Association from which the AssociationEnd is being retrieved.

```
theClass As RoseRT.Class
```

The Class whose AssociationEnd is being returned.

GetOtherAssociationEnd (Classifier : Classifier) : AssociationEnd

Description

Retrieves an AssociationEnd associated with a specified class.

Syntax

```
Set theAssociationEnd = theAssociation.GetOtherAssociationEnd
(theClass)
```

```
theAssociationEnd As RoseRT.AssociationEnd
```

Returns the AssociationEnd that corresponds to the specified class.

```
theAssociationEnd As RoseRT.AssociationEnd
```

Association from which the AssociationEnd is being retrieved.

```
theClass As RoseRT.Class
```

Class whose AssociationEnd is being returned.

NameIsDirectional () : Boolean

Description

Checks whether the association has a name directional AssociationEnd setting.

Syntax

```
IsDirectional = theAssociation.NameIsDirectional ()
```

```
IsDirectional As Boolean
```

Returns a value of True is the association has a name directional setting.

```
theAssociation As RoseRT.Association
```

Association whose name direction setting is being checked.

SetAssociationEndForNameDirection (theAssociationEnd : AssociationEnd) :

Description

Sets the AssociationEnd that is the name direction for the association.

Syntax

```
theAssociation.SetAssociationEndForNameDirection theAssociationEnd
```

```
theAssociation As RoseRT.Association
```

Association whose name direction AssociationEnd is being set.

```
theAssociationEnd As RoseRT.AssociationEnd
```

AssociationEnd being set as the association's name direction.

SetEnds (End1 : ModelElement, End2 : ModelElement) : Boolean

Description

Sets the ends of an Association.

Syntax

```
EndSets = theAssociation.SetEnds( End1, End2 )
```

```
EndSets As Boolean
```

Returns a value of True when ends are set successfully.

```
theAssociationAs RoseRT.Association
```

Association to which the Ends are being set.

```
End1 As RoseRT.ModelElement
```

Model Element at first end of the Association.

```
End2 As RoseRT.ModelElement
```

Model Element at second end of the Association.

AssociationEnd

Description

AssociationEnds denote the purpose or capacity in which one class associates with another. The AssociationEnd class exposes a set of attributes and operations that

- Determine the characteristics of AssociationEnd
- Allow you to retrieve AssociationEnds from a model

Check the lists of attributes and operations for complete information.

Derived from Relation

Public Attributes

Aggregate : Boolean

Description

Indicates whether the AssociationEnd is an aggregate class.

AssociateModelElement : ModelElement

Description

Model Element belonging to the AssociationEnd.

Association : Association

Description

Specifies an association belonging to the AssociationEnd.

Classifier : Classifier

Description

Model Element belonging to the AssociationEnd, casted as a Classifier. Nothing gets returned if the Associate Model Element is not a Classifier.

Constraints : String

Description

Specifies any constraints (expressions of semantic conditions that must be preserved) on the AssociationEnd.

Containment : AssociationEndContainment

Description

The Containment property is a rich data type that controls the containment relationship of an association end.

Friend : Boolean

Description

Indicates whether the AssociationEnd is a Friend, allowing access to its non-public attributes and operations.

Keys : AttributeCollection

Description

Specifies the keys belonging to the AssociationEnd.

Multiplicity : String

Description

Multiplicity of an Association End.

Navigable : Boolean

Description

Indicates whether the AssociationEnd is navigable.

Static : Boolean

Description

Indicates whether the AssociationEnd is static.

UseCase : UseCase

Description

Model Element belonging to the AssociationEnd, casted as a UseCase. Nothing gets returned if the Associate Model Element is not a UseCase.

Visibility : AssociationEndVisibilityKind

Description

The Visibility property is a rich data type that controls access to the Association End object.

Public Operations

AddKey (theName : String, theType : String) : Attribute

Description

Returns a key for an AssociationEnd based on a specified attribute name and type.

Syntax

```
Set theKey = theAssociationEnd.AddKey (theAttrNam, theAttrType)


theKey As RoseRT.Attribute
```

Returns the key as an attribute.

```
theAssociationEnd As RoseRT.AssociationEnd
```

AssociationEndto which the key is being added.

```
theAttrName As String
```

Name of the attribute to use as a key.

```
theAttrType As String
```

Attribute type to use as a key.

DeleteKey (theAttr : Attribute) : Boolean

Description

Deletes a key from an AssociationEnd.

Syntax

```
Deleted = theAssociationEnd.DeleteKey (theAttribute)
```

```
Deleted As Boolean
```

Set to True when the key is deleted.

```
theAssociationEnd As RoseRT.AssociationEnd
```

AssociationEnd from which the key is being deleted.

```
theAttribute As Attribute
```

Name of the attribute whose key is being deleted.

GetClassName () : String

Description

Returns the name of the class belonging to the AssociationEnd.

Syntax

```
theName = theAssociationEnd.GetClassName ( )
```

```
theName As String
```

Returns the name of the class belonging to the AssociationEnd. If the class does not exist, a name other than a class name may be returned by the function.

```
theAssociationEnd As RoseRT.AssociationEnd
```

AssociationEnd whose class name is being retrieved.

IsAssociateClass () : Boolean

Description

Returns whether the Associate Model Element is a Class.

Syntax

```
IsAClass = theAssociationEnd.IsAssociateClass()
```

```
IsAClass As Boolean
```

Returns a value of True if the Associate Model Element is a Class.

```
theAssociationEnd As RoseRT.AssociationEnd
```

The Association End to determine whether the associate Model Element is a Class

AssociationEndContainment

Description

Rich type used to determine how an association end containment attribute.Valid values are defined in RsContainment enumeration.

Derived from RichType

AssociationEndVisibilityKind

Description

Rich type used to determine how an association end can be accessed from other Classifiers. Valid values are defined in RsVisibility enumeration.

Derived from RichType

Classifier Classes

Classifier Classes include

- *Capsule* on page 303
    - Public Attributes

      Structure : CapsuleStructure on page 303
- *Class* on page 304
    - Public Attributes

      ClassKind : ClassKind on page 304

      Concurrency : ClassConcurrency on page 304

      FundamentalType : Boolean on page 304

      Multiplicity : String on page 305

      Parameters : ParameterCollection on page 305

      ParentClass : Class on page 305

      Persistence : Boolean on page 305

      Space : String on page 305
    - Public Operations

      AddInstantiateRel (theRelationName : String, theParentClassName : String) : InstantiateRelation on page 305

      AddNestedClass (theName : String) : Class on page 306

AddParameter (theName : String, theType : String, theDef : String, position : Integer) : Parameter on page 307

DeleteInstantiateRel (theInstantiateRel : InstantiateRelation) : Boolean on page 307

DeleteNestedClass (theClass : Class) : Boolean on page 308

GetInstantiateRelations () : InstantiateRelationCollection on page 308

GetNestedClasses () : ClassCollection on page 309

IsNestedClass () : Boolean on page 309

- *ClassConcurrency* on page 310
- *ClassKind* on page 310
- *Classifier* on page 310
  - Public Attributes

    Abstract : Boolean on page 311

    AssignedLanguage : String on page 311

    Attributes : AttributeCollection on page 311

    Collaborations : CollaborationCollection on page 311

    Operations : OperationCollection on page 312

    ParentLogicalPackage : LogicalPackage on page 312

    StateMachine : StateMachine on page 312

    SystemClass : Boolean on page 313

    Visibility : ClassifierVisibilityKind on page 313

  - Public Operations

    AddAssociation (theSupplierRoleName : String, theSupplierRoleType : String) : Association on page 313

    AddAttribute (theName : String, theType : String, initVal : String) : Attribute on page 314

    AddClassDependency (thSupplierName : String, theSupplierType : String) : ClassDependency on page 315

    AddCollaboration (theCollabName : String) : Collaboration on page 315

AddGeneralization (theRelationName : String, theParentClassName : String) : Generalization on page 316

AddGeneralizationEx (theRelationName : String, theParentClassName : String, ExcludeSuperclassProps : Boolean) : Generalization on page 316

AddOperation (theName : String, retType : String) : Operation on page 317

AddRealizeRel (theRelationName : String, theSupplierName : String) : RealizeRelation on page 318

CreateStateMachine () : on page 318

DeleteAssociation (thAss : Association) : Boolean on page 319

DeleteAttribute (theAttr : Attribute) : Boolean on page 319

DeleteClassDependency (theDependency : ClassDependency) : Boolean on page 320

DeleteCollaboration (theCollab : Collaboration) : Boolean

DeleteGeneralization (theGeneralization : Generalization) : Boolean on page 321

DeleteGeneralizationEx (theGeneralization : Generalization, AbsorbSuperClassProps : Boolean) : Boolean on page 322

DeleteOperation (theOper : Operation) : Boolean on page 322

DeleteRealizeRel (theRel : RealizeRelation) : Boolean on page 323

DeleteStateMachine () : on page 323

GetAssociateAssociationEnds () : AssociationEndCollection on page 324

GetAssociationEnds () : AssociationEndCollection on page 324

GetAssociations () : AssociationCollection

GetClassDependencies () : ClassDependencyCollection on page 325

GetClassifier () : Classifier on page 325

GetGeneralizations () : GeneralizationCollection on page 326

GetRealizeRelations () : RealizeRelationCollection on page 326

GetSubClasses () : ClassifierCollection on page 326

GetSuperClasses () : ClassifierCollection on page 327

- *ClassifierVisibilityKind* on page 327
- *Parameter* on page 328

- Public Attributes

  Const : Boolean on page 328

  InitValue : String on page 328

  Type : String on page 328

- *Protocol* on page 329
  - Public Attributes

    InSignals : SignalCollection on page 329

    Interactions : InteractionCollection on page 329

    OutSignals : SignalCollection on page 329

  - Public Operations

    AddInSignal () : Signal on page 329

    AddInteraction (name : String) : Interaction on page 330

    AddOutSignal () : Signal on page 330

    DeleteInSignal (theSignal : Signal) : Boolean on page 331

    DeleteInteraction (theInteraction : Interaction) : Boolean on page 331

    DeleteOutSignal (theSignal : Signal) : Boolean

- *RsClassKind* on page 332
  - Public Attributes

    rsInstantiatedClass : Integer = 2 on page 332

    rsInstantiatedUtility : Integer = 5 on page 333

    rsMeta : Integer = 6 on page 333

    rsNormalClass : Integer = 0 on page 333

    rsParametrizedClass : Integer = 1 on page 333

    rsParametrizedUtility : Integer = 4 on page 333

    rsUtilityClass : Integer = 3 on page 333

- *RsConcurrency* on page 334
  - Public Attributes

    rsActiveConcurrency : Integer = 2 on page 334

rsGuardedConcurrency : Integer = 1 on page 334

rsSequentialConcurrency : Integer = 0 on page 334

rsSynchronousConcurrency : Integer = 3 on page 334

- *Signal* on page 335
  - Public Attributes

    Class : Class on page 335

    ClassName : String on page 336

    In : Boolean on page 336

    ParentProtocol : Protocol on page 336

Capsule

Description

Capsules are the fundamental modeling element of real-time systems. A capsule represents independent flows of control in a system. Capsules have much of the same properties as classes; for example they can have operations and attributes. Capsules may also participate in dependency, generalization, and association relationships. However they also have several specialized properties which distinguish them from classes.

Derived from Classifier

Public Attributes

Structure : CapsuleStructure

Description

The CapsuleStructure Model Element object that maps to a capsule's Structure Diagram.

Class

Description

The Class class allows you to get and set the characteristics and relationships of specific classes in a model.

Some of the questions answered by class properties are

- Is this an abstract class?
- Is this class a fundamental type?
- Is this class persistent?
- Can this class be concurrent with any other classes?
- What set of attributes and operations belong to this class?
- What relationships are defined between this class and other objects in the model?

Class operations allow you to get and set this information for the classes in the model. Check the lists of attributes and operations for complete information.

Derived from Classifier

Public Attributes

ClassKind : ClassKind

Description

The ClassKind property is a rich data type that determines the type of the class.

Concurrency : ClassConcurrency

Description

The Concurrency property is a rich data type that denotes the semantics in the presence of multiple threads of control.

FundamentalType : Boolean

Description

Defines this class as a fundamental type.

Multiplicity : String

Description

Multiplicity of the Class.

Parameters : ParameterCollection

Description

Used for class of kind “Parameterized Class” or “Parameterized Class Utility”. Formal parameters to be used for their instantiation.

ParentClass : Class

Description

Specifies the parent class of this class.

Persistence : Boolean

Description

Defines the lifetime of the instances of a class. A persistent element is expected to have a life span beyond that of the program or one that is shared with other threads of control or other processes.

Space : String

Description

Defines the space algorithm to use for the class.

Public Operations

AddInstantiateRel (theRelationName : String, theParentClassName : String) : InstantiateRelation

Description

Adds an instantiate relation to a class.

Syntax

```
Set theIntantiateRelation = theClass.AddInstantiateRel(
theRelationName, theParentClassName )
```

```
theIntantiateRelation As RoseRT.InstantiateRelation
```

Returns a new Instantiate Relation denoting theClass as an instantiation of the parametrized class named theParentClassName.

```
theClass As RoseRT.Class
```

The Class to instantiate from the parametrized class whose name is theParentClassName.

```
theRelationName As String
```

The name of the relation.

```
theParentClassName As String
```

Name of the parametrized class that instantiates theClass.

AddNestedClass (theName : String) : Class

Description

Creates a new nested class and adds it to a class.

Syntax

```
Set theNestedClass = theClass.AddNestedClass (theName)
```

```
theNestedClass As RoseRT.Class
```

Returns the nested class being added to the class.

```
theClass As RoseRT.Class
```

Class to which the nested class is being added.

```
theName As String
```

Name of the class being added to the class.

AddParameter (theName : String, theType : String, theDef : String, position : Integer) : Parameter

Description

Adds a formal/actual parameter to a parametrized/instantiated class.

Syntax

```
Set theParameter = theClass.AddParameter( theName, theType, theDef,
position )
```

`theParameter As RoseRT.Parameter`

Returns a new formal/actual Parameter for the parametrized/instantiated class theClass.

`theClass As RoseRT.Class`

The parametrized/instantiated class to add a parameter to.

`theName As String`

The name of the new formal/actual Parameter.

`theType As String`

The type of the new formal/actual Parameter.

`theDef As String`

The default value of the new formal/actual Parameter.

`position As Integer`

The position of the new formal Parameter in the parameter list.

DeleteInstantiateRel (theInstantiateRel : InstantiateRelation) : Boolean

Description

Deletes an instantiate relation from a class.

Syntax

```
IsDeleted = theClass.DeleteInstantiateClass( theInstantiateRel )
```

`IsDeleted As Boolean`

Returns whether theInstantiateRel was deleted successfully from theClass.

`theClass As RoseRT.Class`

The Class to delete an Instantiate Relation from.

`theInstantiateRel As RoseRT.InstantiateRelation`

The relation to delete.

DeleteNestedClass (theClass : Class) : Boolean

Description

Deletes an association from a class.

Syntax

```
Deleted = theClass.DeleteNestedClass (theNestedClass)
```

`Deleted As Boolean`

Returns a value of True when the nested class is deleted.

`theClass As RoseRT.Class`

Class from which the nested class is being deleted.

`theNestedClass As RoseRT.Class`

Nested class being deleted.

GetInstantiateRelations () : InstantiateRelationCollection

Description

Returns the collection of Instantiate Relations that belong to a class.

Syntax

```
Set theInstantiateRelations = theClass.GetInstantiateRelations()
```

```
theInstantiateRelations As RoseRT.InstantiateRelationCollection
```

Returns the collection of Instantiate Relations that belong to a theClass.

```
theClass As RoseRT.Class
```

The Class to return Instantiate Relation Collection from.

GetNestedClasses () : ClassCollection

Description

Retrieves the nested class collection from a class and returns it in the specified object.

Syntax

```
Set theNestedClasses = theClass.GetNestedClasses ( )
```

```
theNestedClasses As RoseRT.ClassCollection
```

Returns the nested class collection from the class.

```
theClass As RoseRT.Class
```

Class from which the collection is being retrieved.

IsNestedClass () : Boolean

Description

Determines whether a class is nested.

Syntax

```
IsNested = theClass.IsNestedClass ( )
```

```
IsNested As Boolean
```

Returns a value of True if the specified class is nested.

`theClass As RoseRT.Class`

The instance of the class being checked for nesting.

ClassConcurrency

Description

Rich type used to determine concurrency of an operation or of a Class.

Valid values are defined in RsConcurrency enumeration.

Derived from RichType

ClassKind

Description

Rich type used to determine kind of a Class. Valid values are defined in RsClassKind enumeration.

Derived from RichType

Classifier

Description

A classifier is a base class that describes behavioral and structural features (attributes and operations).

Derived from ModelElement

Public Attributes

Abstract : Boolean

Description

Indicates whether the classifier is an abstract classifier.

Syntax

```
Classifier.Abstract
```

Property Type:

Boolean

AssignedLanguage : String

Description

The implementation language for the classifier from the available languages. The analysis selection indicates that no code will be generated for the classifier.

Attributes : AttributeCollection

Description

Causes the classifier to inherit all of the attributes of a specified attribute collection.

Syntax

```
Classifier.Attributes
```

Property Type:

AttributeCollection

Collaborations : CollaborationCollection

Description

Collaborations that belong to this classifier.

Operations : OperationCollection

Description

Causes the classifier to inherit all of the operations of a specified operation collection.

Syntax

```
Classifier.Operations
```

Property Type:

OperationsCollection

ParentLogicalPackage : LogicalPackage

Description

Indicates the LogicalPackage that contains the classifier.

Syntax

```
Classifier.ParentLogicalPackage
```

Property Type

LogicalPackage

StateMachine : StateMachine

Description

Specifies the state machine that belongs to the classifier. A state machine defines all of the state information, including states, transitions, and state diagrams, defined for a given classifier.

A classifier can have zero or one state machine.

Syntax

```
Classifier.StateMachine
```

Property Type:

StateMachine

SystemClass : Boolean

Description

Determines whether a class is a system class.

Examples of system classes are

- Exception
- Frame
- Log
- Timing

Visibility : ClassifierVisibilityKind

Description

The Visibility property is a RichType that specifies how a classifier and its elements are viewed outside of the defined package.

Public Operations

AddAssociation (theSupplierRoleName : String, theSupplierRoleType : String) : Association

Description

Adds an association to a classifier and returns it in the specified object.

Syntax

```
Set theAssociation = theClassifier.AddAssociation
(theSupplierRoleName, theSupplierRoleType)
```

```
theAssociation As RoseRT.Association
```

Returns the association being added to the class.

```
theClassifier As RoseRT.Class
```

Classifier to which the association is being added.

```
theSupplierRoleName As String
```

Name of the supplier role in the association.

```
theSupplierRoleType As String
```

Type of the supplier role in the association.

AddAttribute (theName : String, theType : String, initVal : String) : Attribute

Description

Creates a new attribute and adds it to a classifier.

Syntax

```
Set theAttribute = theClassifier.AddAttribute (AttName, AttrType,
InitValue)
```

```
theAttribute As RoseRT.Attribute
```

Returns the attribute being added to the classifier.

```
theClassifier As RoseRT.Class
```

Classifier to which the attribute is being added.

```
AttName As String
```

Name of the attribute being added to the classifier.

```
AttrType As String
```

Type of attribute being added to the classifier.

```
InitValue As String
```

Initial value of the attribute.

AddClassDependency (thSupplierName : String, theSupplierType : String) : ClassDependency

Description

Creates a new class dependency and adds it to a class.

Syntax

```
Set theDependency = theClass.AddClassDependency (theSupplierName,
theSupplierType)
```

`theClassDependency As ClassDependency`

Returns the class dependency being added to the class.

`theClass As Class`

Class to which the class dependency is being added.

`theSupplierName As String`

Name of the supplier class of the class dependency.

`theSupplierType As String`

Type of supplier of the class dependency.

AddCollaboration (theCollabName : String) : Collaboration

Description

Adds a collaboration to a classifier and returns it in the specified object.

Syntax

```
Set theCollaboration = theClassifier.AddCollaboration( theCollabName )
```

`theCollaboration As RoseRT.Collaboration`

Returns the Collaboration being added to the classifier.

`theClassifier As RoseRT.Classifier`

Classifier to which the collaboration is being added.

```
theCollabName As String
```

Name of the new Collaboration.

AddGeneralization (theRelationName : String, theParentClassName : String) : Generalization

Description

Adds a Generalization relationship to a classifier and returns it in the specified object.

Syntax

```
Set theGeneralization = theClassifier.AddGeneralization(
theRelationName, theParentClassifierName )
```

```
theGeneralization As RoseRT.Generalization
```

Returns the Generalization being added to the classifier.

```
theClassifier As RoseRT.Classifier
```

Classifier to which the Generalization is being added.

```
theRelationName As String
```

Name of the new Generalization.

```
theParentClassName As String
```

Name of the parent classifier in the Generalize relationship.

AddGeneralizationEx (theRelationName : String, theParentClassName : String, ExcludeSuperclassProps : Boolean) : Generalization

Description

Adds a Generalization relationship to a classifier and returns it in the specified object.

Syntax

```
Set theGeneralization = theClassifier.AddGeneralizationEx(
theRelationName, theParentClassifierName, ExcludeSuperclassProps )
```

`theGeneralization As RoseRT.Generalization`

Returns the Generalization being added to the classifier.

`theClassifier As RoseRT.Classifier`

Classifier to which the Generalization is being added.

`theRelationName As String`

Name of the new Generalization.

`theParentClassName As String`

Name of the parent classifier in the Generalize relationship.

`ExcludeSuperclassProps As Boolean`

Determines whether to exclude the new superclass' properties. Only meaningful for Capsule and Protocol derived classes.

AddOperation (theName : String, retType : String) : Operation

Description

Creates a new operation and adds it to a classifier.

Syntax

```
Set theOperation = theClassifier.AddOperation (OperationName,
OperationType)
```

`theOperation As RoseRT.Operation`

Returns the operation being added to the class.

`theClass As RoseRT.Classifier`

Classifier to which the operation is being added.

```
OperationName As String
```

Name of the operation being added to the classifier.

```
OperationType As String
```

Type of operation being added to the classifier.

AddRealizeRel (theRelationName : String, theSupplierName : String) : RealizeRelation

Description

Creates a new realize relation and adds it to a classifier.

Syntax

```
Set theRealizeRelation = theClassifier.AddRealizeRel (theRelationName,
theInterfaceName)
```

```
theRealizeRelation As RoseRT.RealizeRelation
```

Returns the realize relation being added to the class.

```
theClassifier As RoseRT.Classifier
```

Classifier to which the realize relation is being added.

```
theRelationName As String
```

Name of the relation being added.

```
theInterfaceName As String
```

Name of the interface with which to create the realize relation.

CreateStateMachine () :

Description

Creates a state machine for a classifier.

Note: A classifier can have zero or one state machine. Multiple state machines are not allowed.

Syntax

```
theClassifier.CreateStateMachine
```

```
theClassifier As RoseRT.Classifier
```

Classifier to which you are adding the state machine.

DeleteAssociation (thAss : Association) : Boolean

Description

Deletes an association from a classifier.

Syntax

```
Deleted = theClassifier.DeleteAssociation (theAssociation)
```

```
Deleted As Boolean
```

Returns a value of True when the association is deleted.

```
theClassifier As RoseRT.Classifier
```

Class from which the association is being deleted.

```
theAssociation As RoseRT.Association
```

Name of the association being deleted. (The association must belong to the specified classifier.)

DeleteAttribute (theAttr : Attribute) : Boolean

Description

Deletes an attribute from a classifier.

Syntax

```
Deleted = theClassifier.DeleteAttribute (theAttribute)
```

```
Deleted As Boolean
```

Returns a value of True when the attribute is deleted.

```
theClassifier As RoseRT.Classifier
```

Classifier from which the attribute is being deleted.

```
theAttribute As RoseRT.Attribute
```

Attribute being deleted from the classifier.

DeleteClassDependency (theDependency : ClassDependency) : Boolean

Description

Deletes a classifier dependency from a classifier.

Syntax

```
IsDeleted = theClassifier.DeleteClassifierDependency (theDependency)
```

```
IsDeleted As Boolean
```

Returns a value of True when the classifier dependency is deleted.

```
theClassifier As RoseRT.Classifier
```

Classifier from which the classifier dependency is being deleted.

```
theDependency As RoseRT.ClassifierDependency
```

Classifier dependency being deleted.

DeleteCollaboration (theCollab : Collaboration) : Boolean

Description

Deletes a collaboration from a classifier.

Syntax

```
Deleted = theClassifier.DeleteCollaboration( theCollab )
```

```
Deleted As Boolean
```

Returns a value of True when the collaboration gets deleted successfully from the classifier.

```
theClassifier As RoseRT.Classifier
```

Classifier from which the collaboration is being deleted.

```
theCollab As RoseRT.Collaboration
```

The collaboration being deleted.

DeleteGeneralization (theGeneralization : Generalization) : Boolean

Description

Deletes a Generalization relation from a classifier.

Syntax

```
Deleted = theClassifier.DeleteGeneralization( theGeneralization )
```

```
Deleted As Boolean
```

Returns a value of True when the generalization gets deleted successfully from the classifier.

```
theClassifier As RoseRT.Classifier
```

Classifier from which the generalization is being deleted.

```
theGeneralization As RoseRT.Generalization
```

The generalization being deleted.

DeleteGeneralizationEx (theGeneralization : Generalization, AbsorbSuperClassProps : Boolean) : Boolean

Description

Deletes a Generalization relation from a classifier.

Syntax

```
Deleted = theClassifier.DeleteGeneralizationEx( theGeneralization,
AbsorbSuperClassProps )
```

`Deleted As Boolean`

Returns a value of True when the generalization gets deleted successfully from the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the generalization is being deleted.

`theGeneralization As RoseRT.Generalization`

The generalization being deleted.

`AbsorbSuperClassProps As Boolean`

Determines whether to absorb all of the superclass' properties. Only meaningful for Capsule and Protocol derived classes.

DeleteOperation (theOper : Operation) : Boolean

Description

Deletes an operation from a classifier.

Syntax

```
Deleted = theClassifier.DeleteOperation (theOperation)
```

`Deleted As Boolean`

Returns a value of True when the operation is deleted from the classifier.

```
theClassifier As RoseRT.Classifier
```
Classifier from which the operation is being deleted.

```
theOperation As RoseRT.Operation
```
Operation being deleted from the classifier.

DeleteRealizeRel (theRel : RealizeRelation) : Boolean

Description

Deletes a realize relation from a classifier.

Syntax

```
IsDeleted = theClassifier.DeleteRealizeRel (theRealizeRel)
```

```
IsDeleted As Boolean
```
Returns a value of True relation being added to the classifier.

```
theClassifier As RoseRT.Classifier
```
Classifier from which the realize relation is being deleted.

```
theRealizeRel As RoseRT.RealizeRelation
```
Realize relation being deleted.

DeleteStateMachine () :

Description

Deletes a classifier's state machine from the model.

Syntax

```
theClassifier.DeleteStateMachine
```

```
theClassifier As RoseRT.Classifier
```
Classifier whose state machine is being deleted.

GetAssociateAssociationEnds () : AssociationEndCollection

Description

Retrieves an associate AssociationEnd collection from a classifier and returns it in the specified object.

Syntax

```
Set theAssocAssociationEnd = theClassifier.GetAssociateAssociationEnd
( )
```

`theAssocAssociationEnd As AssocAssociationEndCollection`

Returns the associate AssociationEnd collection from the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the collection is being retrieved.

GetAssociationEnds () : AssociationEndCollection

Description

Retrieves an AssociationEndCollection from a classifier and returns it in the specified object.

Syntax

```
Set theAssociationEnd = theClassifier.GetAssociationEnds ( )
```

`theAssociationEnds As RoseRT.AssociationEndCollection`

Returns the AssociationEndCollection from the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the collection is being retrieved.

GetAssociations () : AssociationCollection

Description

Retrieves an association collection from a classifier and returns it in the specified object.

Syntax

```
Set theAssociationCollection = theClassifier.GetAssociations ( )
```

`theAssociationCollection As RoseRT.AssociationCollection`

Returns the association collection from the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the collection is being retrieved.

GetClassDependencies () : ClassDependencyCollection

Description

Retrieves the classifier dependencies belonging to the classifier.

Syntax

```
Set theClassifierDependencies = theClassifier.GetUsesRelations ( )
```

`theClassifierDependencies As RoseRT.ClassifierDependencyCollection`

Returns the classifier dependency collection belonging to the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the dependencies are being retrieved.

GetClassifier () : Classifier

Description

Returns self as a Classifier.

Syntax

```
Set theClassifier = theClassifier.GetClassifier()
```

`theClassifier As RoseRT.Classifier`

Returns self as a Classifier.

```
theClassifier As RoseRT.Classifier
```

Classifier to remove self as a classifier.

GetGeneralizations () : GeneralizationCollection

Description

Returns the set of Generalization a Classifier is client of.

Syntax

```
Set Generalizations = theClassifier.GetGeneralizations()
```

```
Generalizations As RoseRT.GeneralizationCollection
```

The collection of all Generalization relationships the Classifier is client of.

```
theClassifier As RoseRT.Classifier
```

The classifier to return Generalization it is client of.

GetRealizeRelations () : RealizeRelationCollection

Description

Retrieves the collection of realize relations belonging to the classifier.

Syntax

```
Set theRealizesRelations = theClassifier.GetRealizeRelations ()
```

```
theRealizesRelations As RoseRT.RealizeRelationsCollection
```

Returns the collection of realize relations belonging the classifier.

```
theClassifier As RoseRT.Classifier
```

Classifier from which the collection is being retrieved.

GetSubClasses () : ClassifierCollection

Description

Retrieves the subclasses belonging to the classifier.

Syntax

```
Set theSubclasses = theClassifier.GetSubclasses ( )
```

`theSubclasses As RoseRT.ClassifierCollection`

Returns the collection of classes belonging to the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the collection is being retrieved.

GetSuperClasses () : ClassifierCollection

Description

Retrieves the superclasses belonging to the classifier.

Syntax

```
Set theSuperClassifiers = theClassifier.GetSuperClassifiers ( )
```

`theSuperclassifiers As RoseRT.ClassifierCollection`

Returns the collection of superclassifiers belonging to the classifier.

`theClassifier As RoseRT.Classifier`

Classifier from which the collection is being retrieved.

ClassifierVisibilityKind

Description

Rich type used to determine how a Classifier can be accessed from other Classifiers. Valid values are defined in RsVisibility enumeration.

Derived from RichType

Parameter

Description

Parameters further qualify the behavior of an operation. The parameter class exposes a set of attributes and operations that

- Determine the parameter characteristics such as type and initial value
- Allow you to retrieve parameters

Check the lists of attributes and operations for complete information.

Derived from ModelElement

Public Attributes

Const : Boolean

Description

Indicates that the parameter is a constant

InitValue : String

Description

Indicates the initial value of the parameter object.

Type : String

Description

Indicates the data type of the parameter object

Protocol

Description

Represents the set of messages exchanged between two objects in order to conform to some communication pattern.

Derived from Classifier

Public Attributes

InSignals : SignalCollection

Description

The collection of in signals described by a protocol.

Interactions : InteractionCollection

Description

The collection of interactions describing a protocol.

OutSignals : SignalCollection

Description

The collection of out signals described by a protocol.

Public Operations

AddInSignal () : Signal

Description

Adds an in signal to a protocol and returns it in the specified object.

Syntax

```
Set theSignal = theProtocol.AddInSignal()


theSignal As RoseRT.Signal
```

Returns the in signal being added to the protocol.

```
theProtocol As RoseRT.Protocol
```

Protocol to which the in signal is being added.

AddInteraction (name : String) : Interaction

Description

This function adds an interaction to a protocol and returns it in the specified object.

Syntax

```
Set theInteraction = theProtocol.AddInteraction( name )
```

```
theInteraction As RoseRT.Interaction
```

Returns the interaction being added to the protocol.

```
theProtocol As RoseRT.Protocol
```

Protocol to which the interaction is being added.

```
name As String
```

Name of the interaction to add to the protocol.

AddOutSignal () : Signal

Description

Adds an out signal to a protocol and returns it in the specified object.

Syntax

```
Set theSignal = theProtocol.AddOutSignal()
```

```
theSignal As RoseRT.Signal
```

Returns the out signal being added to the protocol.

```
theProtocol As RoseRT.Protocol
```

Protocol to which the out signal is being added.

DeleteInSignal (theSignal : Signal) : Boolean

Description

Deletes an in signal from a protocol.

Syntax

```
Deleted = theProtocol.DeleteInSignal( theSignal )
```

`Deleted As Boolean`

Returns a value of True when the in signal is successfully deleted from the protocol.

`theProtocol As RoseRT.Protocol`

Protocol to which the in signal is being deleted.

`theSignal As RoseRT.Signal`

The in signal being deleted.

DeleteInteraction (theInteraction : Interaction) : Boolean

Description

Deletes an interaction from a protocol.

Syntax

```
Deleted = theProtocol.DeleteInteraction( theInteraction )
```

`Deleted As Boolean`

Returns a value of True when the interaction is successfully deleted from the protocol.

`theProtocol As RoseRT.Protocol`

Protocol to which the interaction is being deleted.

`theInteraction As RoseRT.Interaction`

The interaction being deleted.

DeleteOutSignal (theSignal : Signal) : Boolean

Description

Deletes an out signal from a protocol.

Syntax

```
Deleted = theProtocol.DeleteOutSignal( theSignal )
```

`Deleted As Boolean`

Returns a value of True when the out signal is successfully deleted from the protocol.

`theProtocol As RoseRT.Protocol`

Protocol to which the out signal is being deleted.

`theSignal As RoseRT.Signal`

The out signal being deleted.

RsClassKind

Description

Enumeration used to set the Value property of the ClassKind Rich Type.

Public Attributes

rsInstantiatedClass : Integer = 2

Description

Class formed from a parameterized class by supplying actual values for parameters.

rsInstantiatedUtility : Integer = 5

Description

Utility class formed from a parameterized class by supplying actual values for parameters.

rsMeta : Integer = 6

Description

Class which describes or is used to instantiate classes instead of objects.

rsNormalClass : Integer = 0

Description

Design-time specification for one or more distinct objects with common structure, attributes, and common behavior, operations.

rsParametrizedClass : Integer = 1

Description

Template for creating any number of instantiated classes that follow its format. A parameterized class declares formal parameters.

rsParametrizedUtility : Integer = 4

Description

Template for creating any number of instantiated utility classes that follow its format. A parameterized class declares formal parameters.

rsUtilityClass : Integer = 3

Description

Specifies a class whose attributes and operations are all class scoped. An instantiated utility class represents an instance of a utility class.

RsConcurrency

Description

Enumeration used to set the Value property of the ClassConcurrency and of the OperationConcurrency Rich Types.

Public Attributes

rsActiveConcurrency : Integer = 2

Description

The class has its own thread of control.

rsGuardedConcurrency : Integer = 1

Description

The semantics of the class are guaranteed in the presence of multiple threads of control. A guarded class requires collaboration among client threads to achieve mutual exclusion.

rsSequentialConcurrency : Integer = 0

Description

The semantics of the class are guaranteed only in the presence of a single thread of control. Only one thread of control can be executing in the method at any one time.

rsSynchronousConcurrency : Integer = 3

Description

The semantics of the class are guaranteed in the presence of multiple threads of control; mutual exclusion is supplied by the class.

RsChangeable

Description

Enumeration used to set the **Value** property of the **Changeable RichType**.

Public Attributes

rsChangeableChangeableKind : Integer = 0

Description

Specifies that the attribute can be modified.

rsFrozenChangeableKind : Integer = 1

Description

Specifies that the attribute cannot be modified.

rsAddOnlyChangeableKind : Integer = 2

Description

Specifies that the attribute can only be updated. For example, items in an array can be appended to, not replaced.

Note: This options is not enforceable in most programming languages.

Signal

Description

A signal is a specification of an asynchronous stimulus communicated between instances.

Derived from ModelElement

Public Attributes

Class : Class

Description

Specifies the class of the data object that is expected as a payload of the message.

ClassName : String

Description

Specifies the classname of the data object that is expected as a payload of the message.

In : Boolean

Description

Specifies whether the signal is an in signal.

ParentProtocol : Protocol

Description

Protocol that own the signal.

Feature Classes

Feature Classes include

- *Attribute* on page 338
    - Public Attributes

      Containment : AttributeContainment on page 338

      Derived : Boolean on page 338

      InitValue : String on page 339

      OwnerScope : OwnerScope on page 339

      ParentClassifier : Classifier on page 339

      Type : String on page 339

      Visibility : AttributeVisibilityKind on page 339
- *AttributeContainment* on page 340
- *AttributeVisibilityKind* on page 340
- *Operation* on page 340
    - Public Attributes

      Abstract : Boolean on page 341

      Code : String on page 341

Concurrency : OperationConcurrency on page 341

Exceptions : String on page 341

OwnerScope : OwnerScope on page 341

Parameters : ParameterCollection on page 341

ParentClassifier : Classifier on page 342

Postconditions : String on page 342

Preconditions : String on page 342

Protocol : String on page 342

Qualification : String on page 342

Query : Boolean on page 342

ReturnType : String on page 342

Semantics : String on page 343

Size : String on page 343

Time : String on page 343

Virtual : Boolean on page 343

Visibility : OperationVisibilityKind on page 343

- Public Operations

AddParameter (theName : String, theType : String, theDef : String, position : Integer) : Parameter on page 343

DeleteParameter (theParameter : Parameter) : Boolean on page 344

RemoveAllParameters () : on page 345

- *OperationConcurrency* on page 345
- *OperationVisibilityKind* on page 345
- *OwnerScope* on page 346
- *RsOwnerScope* on page 346

- Public Attributes

rsClassifierScopeKind : Integer = 1 on page 346

rsInstanceScopeKind : Integer = 0 on page 346

Attribute

Description

Attributes define the characteristics of a class. Each object in a classifier has the same attributes, but the values of the attributes may be different.

The attribute class exposes a set of attributes and operations that determine the characteristics of these attributes and that allow you to retrieve them from a model.

Some of the characteristics determined by attribute class properties are

- Type
- Initial value
- Whether the attribute is static; whether it is derived
- Attribute visibility

Check the lists of attributes and operations for complete information.

Derived from ModelElement

Public Attributes

Containment : AttributeContainment

Description

The Containment property is a rich data type that controls the containment relationship of an attribute.

Derived : Boolean

Description

Indicates whether the attribute is derived.

Changeability : Changeability

The **Changeable** property is a **RichType** that specifies the manner in which you can modify an attribute. The options available are:

- **Changeable** - The attribute can be modified.
- **Frozen** - The attribute cannot be modified.

- **Add-only** - The attribute can only be updated. For example, items in an array can be appended to, not replaced.

  Note: This options is not enforceable in most programming languages.

Example

```
Dim changeability As RoseRT.RichType
Set changeability = myAttribute.Changeable
changeability.Value = RsFrozenChangeableKind
```

For additional information on the possible values, see *RsChangeable* on page 334.

InitValue : String

Description

Indicates the initial value of the attribute object.

OwnerScope : OwnerScope

Description

The **OwnerScope** property is a **RichType** that determines whether a single instance of the attribute is shared for all instances of the classifier or if each instance of the class have a separate attribute instance.

ParentClassifier : Classifier

Description

Specifies the Classifier to which the attribute belongs.

Type : String

Description

Indicates the data type of the attribute object.

Visibility : AttributeVisibilityKind

Description

The Visibility property is a RichType that determines how an attribute can be accessed from other classifiers.

AttributeContainment

Description

Rich type used to determine the containment of an attribute within a Classifier. Valid values are defined in RsContainment enumeration.

Derived from RichType

AttributeVisibilityKind

Description

Rich type used to determine the visibility of an attribute within a Classifier. Valid values are defined in RsVisibilityKind enumeration.

Derived from RichType

Operation

Description

Objects in a class carry out their defined responsibilities by using operations. Each operation performs a single, cohesive function. The operation classifier exposes a set of attributes and operations that

- Determine operation characteristics
- Add or remove parameters from operations
- Allow you to retrieve operations

Check the lists of attributes and operations for complete information.

Derived from ModelElement

Public Attributes

Abstract : Boolean

Description

Indicates that the operation is an abstract definition that should be overridden by specific implementations in subclasses.

Code : String

Description

Detailed implementation code for the operation.

Concurrency : OperationConcurrency

Description

The Operation Concurrency property is a rich data type that denotes the semantics in the presence of multiple threads of control.

Exceptions : String

Description

Identifies the set of exceptions that can be raised by an operation.

OwnerScope : OwnerScope

Description:

The OwnerScope property is a RichType that determines whether an operation is scoped as a class operation or whether it is an instance operation.

Parameters : ParameterCollection

Description

Defines the collection of parameters that is valid for the operation.

ParentClassifier : Classifier

Description

Specifies the classifier to which the operation belongs.

Postconditions : String

Description

Controls invariants that are satisfied by the operation; that is, the exit behavior of the operation.

Preconditions : String

Description

Controls invariants assumed by the operation; that is, the entry behavior of an operation.

Protocol : String

Description

Specifies the set of operations that a client may perform on an object and the legal order in which the operations can be called.

Qualification : String

Description

Identifies language-specific features used to qualify an operation.

Query : Boolean

Description

Indicates that the operation is read-only and does not modify the object's state.

ReturnType : String

Description

Determines the object type to be returned by an operation; can be set to any valid data type, rich data type, or object type.

Semantics : String

Description

Controls the action of an operation.

Size : String

Description

Identifies the relative or absolute amount of storage used when the operation is called.

Time : String

Description

Identifies the relative or absolute amount of time required to complete the operation.

Virtual : Boolean

Description

Indicates whether the operation is virtual

Visibility : OperationVisibilityKind

Description

The Visibility property is a RichType that determines how an operation can be accessed from other classifiers.

Public Operations

AddParameter (theName : String, theType : String, theDef : String, position : Integer) : Parameter

Description

Creates a new parameter and adds it to an operation.

Syntax

```
Set theParameter = theOperation.AddParameter (ParameterName,
ParameterType, InitValue, Position)


theParameter As RoseRT.Parameter
```

Returns the parameter being added to the operation.

```
theOperation As RoseRT.Operation
```

Operation to which the parameter is being added.

```
ParameterName As String
```

Name of the parameter being added to the operation.

```
ParameterType As String
```

Type of parameter being added to the operation.

```
InitValue As String
```

Initial value of the added parameter.

```
Position As Integer
```

Order of the parameter in the operation's parameter list.

DeleteParameter (theParameter : Parameter) : Boolean

Description

Deletes a parameter from an operation.

Syntax

```
Deleted = theOperation.DeleteParameter (theParameter)
```

```
Deleted As Boolean
```

Returns a value of True when the specified parameter is deleted from the operation.

```
theOperation As RoseRT.Operation
```

Operation from which the parameter is being deleted.

```
theParameter As RoseRT.Parameter
```

Parameter being deleted from the operation.

RemoveAllParameters () :

Description

Removes all parameters from an operation.

Syntax

```
theOperation.RemoveAllParameters


theOperation As RoseRT.Operation
```

Operation from which the parameters are being removed.

OperationConcurrency

Description

Rich type used to determine the concurrency of an operation within a Classifier. Valid values are defined in RsConcurrency enumeration.

Derived from RichType

OperationVisibilityKind

Description

Rich type used to determine the visibility of an operation within a Classifier. Valid values are defined in RsVisibilityKind enumeration.

Derived from RichType

OwnerScope

Description

Rich type used to determine the scope of an attribute within a Classifier. Valid values are defined in RsOwnerScope enumeration.

Derived from RichType

RsOwnerScope

Description

Enumeration used to set the Value property of the OwnerScope Rich Type.

Public Attributes

rsClassifierScopeKind : Integer = 1

Description

There is a single instance of the attribute for all instances of the class (a static member in C++ terminology).

rsInstanceScopeKind : Integer = 0

Description

Each instance of the class will have a separate attribute instance.

Collaboration Classes

Collaboration classes include

- *AssociationEndRole* on page 350
  - Public Attributes

    AssociationRole : AssociationRole on page 351

    Base : AssociationEnd on page 351
- *AssociationRole* on page 351
  - Public Attributes

    Base : Association on page 351

    BaseName : String on page 351

    Multiplicity : String on page 351

    ParentCollaboration : Collaboration on page 352
- *CapsuleRole* on page 352
  - Public Attributes

    Capsule : Capsule on page 352

    Cardinality : String on page 352

    Genericity : Genericity on page 352

    PortRoles : PortRoleCollection on page 352

    Substitutable : Boolean on page 353
- *CapsuleStructure* on page 353
  - Public Attributes

    Ports : PortCollection on page 353
  - Public Operations

    AddCapsuleRole (capsuleName : String) : CapsuleRole on page 353

    AddPort (name : String, protocolName : String) : Port on page 354

    CopyToCollaboration (toContext : ModelElement, fromContext : ModelElement) : Collaboration on page 354

    DeleteCapsuleRole (role : CapsuleRole) : Boolean on page 355

DeletePort (port : Port) : Boolean on page 356

- *ClassifierRole* on page 356
  - Public Attributes

    Classifier : Classifier on page 356

    ClassifierName : String on page 357

    Multiplicity : String on page 357

    ParentCollaboration : Collaboration on page 357
  - Public Operations

    ClassifierRole () : ClassifierRole on page 357
- *Collaboration* on page 358
  - Public Attributes

    AssociationRoles : AssociationRoleCollection on page 358

    ClassifierRoles : ClassifierRoleCollection on page 358

    Connectors : ConnectorCollection on page 358

    Diagram : CollaborationDiagram on page 358

    Interactions : InteractionCollection on page 358

    ParentClassifier : Classifier on page 359

    ParentLogicalPackage : LogicalPackage on page 359
  - Public Operations

    AddAssociationRole () : AssociationRole on page 359

    AddCapsuleRole (capsuleName : String) : CapsuleRole on page 353

    AddClassifierRole () : ClassifierRole on page 360

    AddConnector () : Connector on page 360

    AddInteraction (name : String) : Interaction on page 361

    DeleteAssociationRole (role : AssociationRole) : Boolean on page 361

    DeleteCapsuleRole (role : CapsuleRole) : Boolean on page 355

    DeleteClassifierRole (role : ClassifierRole) : Boolean on page 362

    DeleteConnector (connector : Connector) : Boolean on page 363

DeleteInteraction (interaction : Interaction) : Boolean on page 363

GetLocalInteractions (classifierContext : Classifier) : InteractionCollection on page 364

- *Connector* on page 364
  - Public Attributes

    Cardinality : String on page 364

    Delay : String on page 365

    Port1 : Port on page 365

    Port2 : Port on page 365

    PortRole1 : PortRole on page 365

    PortRole2 : PortRole on page 365
  - Public Attributes

    SetEnds (End1 : ModelElement, End2 : ModelElement) : Boolean on page 365

    SetEndsByNames (End1Name : String, End2Name : String) : Boolean on page 366
- *Genericity* on page 367
- *Port* on page 367
  - Public Attributes

    Cardinality : String on page 367

    Conjugated : Boolean on page 367

    Notification : Boolean on page 368

    Protocol : Protocol on page 368

    Published : Boolean on page 368

    RegistrationMode : RegistrationMode

    RegistrationString : String on page 368

    Relay : Boolean on page 369

    Visibility : PortVisibilityKind on page 369

    Wired : Boolean on page 369
- *PortRole* on page 369

- Public Attributes

  ParentCapsuleRole : CapsuleRole on page 369

  Port : Port on page 369

- *PortVisibilityKind* on page 370
  - Public Attributes

    rsFixed : Integer = 1 on page 370

    rsOptional : Integer = 2 on page 371

    rsPlugIn : Integer = 3 on page 371
- *RegistrationMode* on page 370
- *RsGenericity* on page 370
  - Public Attributes

    rsFixed : Integer = 1 on page 370

    rsOptional : Integer = 2 on page 371

    rsPlugIn : Integer = 3 on page 371
- *RsRegistrationMode* on page 371
  - Public Attributes

    rsApplication : Integer = 2 on page 371

    rsAutomatic : Integer = 1 on page 371

    rsNoMode : Integer = 0 on page 372

AssociationEndRole

Description

An association-end role specifies an endpoint of an association as used in a collaboration.

Derived from AssociationEnd

Public Attributes

AssociationRole : AssociationRole

Description

AssociationRole the AssociationEndRole is an endpoint of.

Base : AssociationEnd

Description

AssociationEnd the AssociationEndRole is a projection of.

AssociationRole

Description

An association role is a specific usage of an association needed in a collaboration.

Derived from Association

Public Attributes

Base : Association

Description

Association the AssociationRole is a projection of.

BaseName : String

Description

Name of the Association the AssociationRole is a projection of.

Multiplicity : String

Description

The number of Association playing this role in a Collaboration.

ParentCollaboration : Collaboration

Description

Collaboration that owns the AssociationRole.

CapsuleRole

Description

Represent a specification of the type of capsules that can occupy a particular position in a capsule's collaboration, or structure.

Derived from ClassifierRole

Public Attributes

Capsule : Capsule

Description

Capsule the CapsuleRole is a projection of.

Cardinality : String

Description

The number of Capsule playing this role in a Collaboration.

Genericity : Genericity

Description

Determines the Genericity of the CapsuleRole.

PortRoles : PortRoleCollection

Description

Port Roles of the Capsule role.

Substitutable : Boolean

Description

Determines whether subclasses of the specified capsule role's class can be instantiated into this role.

CapsuleStructure

Description

Specialization of a Collaboration whose communication pattern is owned by a particular capsule and represents the composite structure of it's capsule roles, ports, and connectors.

Derived from Collaboration

Public Attributes

Ports : PortCollection

Description

Ports involved in the communication pattern described by the CapsuleStructure.

Public Operations

AddCapsuleRole (capsuleName : String) : CapsuleRole

Description

Adds a new CapsuleRole into the CapsuleStructure and returns it.

Syntax

```
Set theCapsuleRole = theCapsuleStructure.AddCapsuleRole( capsuleName )
```

```
theCapsuleRole As RoseRT.CapsuleRole
```

Returns the new CapsuleRole added to the CapsuleStructure.

```
theCapsuleStructure As RoseRT.Classifier
```

CapsuleStructure to which the CapsuleRole is being added.

```
capsuleName As String
```

Name of a Capsule the CapsuleRole is a projection of.

AddPort (name : String, protocolName : String) : Port

Description

Adds a new Port into the CapsuleStructure and returns it.

Syntax

```
Set thePort = theCapsuleStructure.AddPort( name , protocolName )
```

```
thePort As RoseRT.CapsuleRole
```

Returns the new Port added to the CapsuleStructure.

```
theCapsuleStructure As RoseRT.Classifier
```

CapsuleStructure to which the Port is being added.

```
name As String
```

Name of the port added to the CapsuleStructure.

```
protocolName As String
```

Protocol class name for the Port.

CopyToCollaboration (toContext : ModelElement, fromContext : ModelElement) : Collaboration

Description:

Copies the CapsuleStructure into a generic Collaboration. Items specific to CapsuleStructure won't be copied over, i.e. Ports.

Syntax:

```
theCollaboration = theCapsuleStructure.CopyToCollaboration(
theToContext, theFromContext )
```

`theCollaboration As RoseRT.Collaboration`

Returns the converted collaboration.

`theCapsuleStructure As RoseRT.Classifier`

CapsuleStructure that is to be copied and converted to a generic Collaboration.

`theToContext As RoseRT.ModelElement`

Owning item of the new converted Collaboration.

`theFromContext As RoseRT.ModelElement`

Owning item of the original CapsuleStructure.

DeleteCapsuleRole (role : CapsuleRole) : Boolean

Description

Deletes a CapsuleRole from the CapsuleStructure.

Syntax

```
Deleted = theCapsuleStructure.DeleteCapsuleRole( role )
```

`Deleted As Boolean`

Returns a value of True when the CapsuleRole is deleted successfully from the CapsuleStructure.

`theCapsuleStructure As RoseRT.Classifier`

CapsuleStructure from which the CapsuleRole is being deleted.

`role As RoseRT.CapsuleRole`

CapsuleRole to delete from the CapsuleStructure.

DeletePort (port : Port) : Boolean

Description

Deletes a Port from the CapsuleStructure.

Syntax

```
Deleted = theCapsuleStructure.DeletePort( port )
```

```
Deleted As Boolean
```

Returns a value of True when the Port is deleted successfully from the CapsuleStructure.

```
theCapsuleStructure As RoseRT.Classifier
```

CapsuleStructure from which the Port is being deleted.

```
port As RoseRT.CapsuleRole
```

Port to delete from the CapsuleStructure.

ClassifierRole

Description

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.

Derived from ModelElement

Public Attributes

Classifier : Classifier

Description

Classifier the ClassifierRole is a projection of.

ClassifierName : String

Description

Name of the Classifier the ClassifierRole is a projection of.

Multiplicity : String

Description

The number of Classifier playing this role in a Collaboration.

ParentCollaboration : Collaboration

Description

Collaboration that owns the ClassifierRole.

Public Operations

ClassifierRole () : ClassifierRole

Description

Returns the ClassifierRole as a ClassifierRole. This is useful for derived classes' instances type casting.

Syntax

```
Set theClassifierRoleRet = theClassifierRole.ClassifierRole()
theClassifierRoleRet As RoseRT.ClassifierRole
```

Returns the ClassifierRole derived class's instance as a ClassifierRole.

```
theClassifierRole As RoseRT.ClassifierRole
```

ClassifierRole to return as a ClassifierRole.

Collaboration

Description

A Collaboration is a Model Element associated with a Collaboration Diagram. It contains the various Model Elements involved in the communication patterns described in the Collaboration Diagram.

Derived from ModelElement

Public Attributes

AssociationRoles : AssociationRoleCollection

Description

AssociationRoles involved in the communication pattern described by the Collaboration.

ClassifierRoles : ClassifierRoleCollection

Description

ClassifierRoles involved in the communication pattern described by the Collaboration.

Connectors : ConnectorCollection

Description

Connectors involved in the communication pattern described by the Collaboration.

Diagram : CollaborationDiagram

Description

Diagram showing the communication patterns described by the Collaboration.

Interactions : InteractionCollection

Description

Interactions involved in the communication pattern described by the Collaboration.

ParentClassifier : Classifier

Description

Classifier owning the Collaboration. Maybe nothing if owned by a Logical Package.

ParentLogicalPackage : LogicalPackage

Description

Logical Package owning the Collaboration. Maybe nothing if owned by a Classifier.

Public Operations

AddAssociationRole () : AssociationRole

Description

Adds a new AssociationRole into the Collaboration and returns it.

Syntax

```
Set theAssociationRole = theCollaboration.AddAssociationRole()
```

```
theAssociationRole As RoseRT.AssociationRole
```

Returns the new AssociationRole added to the Collaboration.

```
theCollaboration As RoseRT.Collaboration
```

Collaboration to which the AssociationRole is being added.

AddCapsuleRole (capsuleName : String) : CapsuleRole (New 09Jun00)

Description:

Adds a new CapsuleRole into the Collaboration and returns it.

Syntax:

```
Set theCapsuleRole = theCollaboration.AddCapsuleRole( capsuleName )
```

```
theCapsuleRole As RoseRT.CapsuleRole
```

Returns the new CapsuleRole added to the CapsuleStructure.

```
theCollaboration As RoseRT.Collaboration
```

Collaboration to which the CapsuleRole is being added.

```
capsuleName As String
```

Name of a Capsule the CapsuleRole is a projection of.

AddClassifierRole () : ClassifierRole

Description

Adds a new ClassifierRole into the Collaboration and returns it.

Syntax

```
Set theClassifierRole = theCollaboration.AddClassifierRole()
```

```
theClassifierRole As RoseRT.ClassifierRole
```

Returns the new ClassifierRole added to the Collaboration.

```
theCollaboration As RoseRT.Collaboration
```

Collaboration to which the ClassifierRole is being added.

AddConnector () : Connector

Description

Adds a new Connector into the Collaboration and returns it.

Syntax

```
Set theConnector = theCollaboration.AddConnector()
```

```
theConnector As RoseRT.Connector
```

Returns the new Connector added to the Collaboration.

```
theCollaboration As RoseRT.Collaboration
```

Collaboration to which the Connector is being added.

AddInteraction (name : String) : Interaction

Description

Adds a new Interaction into the Collaboration and returns it.

Syntax

```
Set theInteraction = theCollaboration.AddInteraction( name )
```

`theInteraction As RoseRT.Interaction`

Returns the new Interaction added to the Collaboration.

`theCollaboration As RoseRT.Collaboration`

Collaboration to which the Connector is being added.

`name As String`

Name of the Interaction to add to the Collaboration.

DeleteAssociationRole (role : AssociationRole) : Boolean

Description

Deletes an AssociationRole from the Collaboration.

Syntax

```
Deleted = theCollaboration.DeleteAssociationRole( role )
```

`Deleted As Boolean`

Returns a value of True when the AssociationRole is deleted successfully from the Collaboration.

`theCollaboration As RoseRT.Collaboration`

Collaboration from which the AssociationRole is being deleted.

`role As RoseRT.AssociationRole`

The AssociationRole to delete from the Collaboration.

DeleteCapsuleRole (role : CapsuleRole) : Boolean (New 09Jun00)

Description:

Deletes a CapsuleRole from the CapsuleStructure.

Syntax:

```
Deleted = theCapsuleStructure.DeleteCapsuleRole( role )
```

`Deleted As Boolean`

Returns a value of True when the CapsuleRole is deleted successfully from the CapsuleStructure.

`theCapsuleStructure As RoseRT.Classifier`

CapsuleStructure from which the CapsuleRole is being deleted.

`role As RoseRT.CapsuleRole`

CapsuleRole to delete from the CapsuleStructure.

DeleteClassifierRole (role : ClassifierRole) : Boolean

Description

Deletes an ClassifierRole from the Collaboration.

Syntax

```
Deleted = theCollaboration.DeleteClassifierRole( role )
```

`Deleted As Boolean`

Returns a value of True when the ClassifierRole is deleted successfully from the Collaboration.

`theCollaboration As RoseRT.Collaboration`

Collaboration from which the ClassifierRole is being deleted.

`role As RoseRT.ClassifierRole`

The ClassifierRole to delete from the Collaboration.

DeleteConnector (connector : Connector) : Boolean

Description

Deletes an Connector from the Collaboration.

Syntax

```
Deleted = theCollaboration.DeleteConnector( connector )
```

`Deleted As Boolean`

Returns a value of True when the Connector is deleted successfully from the Collaboration.

`theCollaboration As RoseRT.Collaboration`

Collaboration from which the Connector is being deleted.

`connector As RoseRT.Connector`

The Connector to delete from the Collaboration.

DeleteInteraction (interaction : Interaction) : Boolean

Description

Deletes an Interaction from the Collaboration.

Syntax

```
Deleted = theCollaboration.DeleteInteraction( interaction)
```

`Deleted As Boolean`

Returns a value of True when the Interaction is deleted successfully from the Collaboration.

`theCollaboration As RoseRT.Collaboration`

Collaboration from which the Interaction is being deleted.

`interaction As RoseRT.Interaction`

The Interaction to delete from the Collaboration.

GetLocalInteractions (classifierContext : Classifier) : InteractionCollection

Description:

Retrieves the interactions local to a specific classifier context.

Syntax:

```
Set theLocalInteractions = theCollaboration.GetLocalInteractions(
classifier )
```

```
theLocalInteractions As RoseRT.InteractionCollection
```

Returns the collection of local interactions in the given classifier context.

```
theCollaborationAs RoseRT.Collaboration
```

Collaboration from which the collection is being retrieved.

```
classifier As RoseRT.Classifier
```

Classifier context which the interaction is local to.

Connector

Description

Connectors capture the key communication relationships between capsule roles.

Derived from ModelElement

Public Attributes

Cardinality : String

Description

Specifies the number of connectors indicated by a connector line.

Delay : String

Description

Specifies a communication delay across a connector.

Port1 : Port

Description

Port at first end of the Connector. Set when the connector is within a CapsuleStructure. Nothing when the connector is within a Collaboration.

Port2 : Port

Description

Port at second end of the Connector. Set when the connector is within a CapsuleStructure. Nothing when the connector is within a Collaboration.

PortRole1 : PortRole

Description

PortRole at first end of the Connector. Set when the connector is within a Collaboration. Nothing when the connector is within a CapsuleStructure.

PortRole2 : PortRole

Description

PortRole at second end of the Connector. Set when the connector is within a Collaboration. Nothing when the connector is within a CapsuleStructure.

Public Operations

SetEnds (End1 : ModelElement, End2 : ModelElement) : Boolean

Description

Sets the ends of a Connector. Ends can be Port in the context of a CapsuleStructure or PortRole in the context of a Collaboration.

Syntax

```
EndSets = theConnector.SetEnds( End1, End2 )
```

`EndSets As Boolean`

Returns a value of True when ends are set successfully.

`theConnector As RoseRT.Connector`

Connector to which the Ends are being set.

`End1 As RoseRT.ModelElement`

Model Element at first end of the Connector.

`End2 As RoseRT.ModelElement`

Model Element at second end of the Connector.

SetEndsByNames (End1Name : String, End2Name : String) : Boolean

Description

Sets the ends of a Connector. Ends can be Port in the context of a CapsuleStructure or PortRole in the context of a Collaboration.

Syntax

```
EndSets = theConnector.SetEndsByNames( End1Name, End2Name )
```

`EndSets As Boolean`

Returns a value of True when ends are set successfully.

`theConnector As RoseRT.Connector`

Connector to which the Ends are being set.

`End1Name As String`

Fully qualified name of Model Element at first end of the Connector.

`End2Name As String`

Fully qualified name of Model Element at second end of the Connector.

Genericity

Description

Rich type used to determine the Genericity of an attribute within a CapsuleRole. Valid values are defined in RsGenericity enumeration.

Derived from RichType

Port

Description

Ports are objects whose purpose is to send and receive messages to and from capsules instances.

Derived from ClassifierRole

Public Attributes

Cardinality : String

Description

Specifies the number of instances of the port that will appear at run-time.

Conjugated : Boolean

Description

A conjugated port is one in which the standard protocol class definition of in and out signals is reversed.

Notification : Boolean

Description

Determines whether the port will receive rtBound and rtUnbound messages from the services library when ports get connected and unconnected.

Note: rtBound is sent at system priority and rtUnbound is sent at background priority.

Protocol : Protocol

Description

Specifies the protocol class to be used for the port.

Published : Boolean

Description

Determines whether the port is published.

RegistrationMode : RegistrationMode

Description

Only used for non-wired ports. Non-wired ports are registered by name with a name service that performs the connection. Connections are made between protected non-wired ports (service clients) and a single public non-wired port (the service provider). If automatic registration is used, the registration name must be supplied in the RegistrationString attribute and the Services Library will register the name at startup. In the case of application registration, the SAP or SPP is registered at run-time by calling a communication service operation, such as RTEndPortRef::registerSAP() and RTEndPortRef::deregisterSAP(), in the detail level code of a capsule.

RegistrationString : String

Description

Name of service that performs the connection. See RegistrationMode attribute.

Relay : Boolean

Description

Determines whether the port is a Relay port. Relay ports cannot be protected, they must be public. If set to False, then the Port is an End port.

Visibility : PortVisibilityKind

Description

The Visibility property is a RichType that determines whether the port is visible outside of the capsule boundary or not.

Wired : Boolean

Description

Determines whether the port is Wired. Wired ports are connected to other wired ports using connectors. Non-wired ports are connected to other non-wired ports by name.

PortRole

Description

A Port role is a specific usage of an port needed in a collaboration.

Derived from ModelElement

Public Attributes

ParentCapsuleRole : CapsuleRole

Description

Capsule role that owns the port role.

Port : Port

Description

Port the PortRole is a projection of.

PortVisibilityKind

Description

Rich type used to determines whether the port is visible outside of the capsule boundary. Valid values are defined in RsVisibilityKind enumeration.

Derived from RichType

RegistrationMode

Description

Rich type used to determine the RegistrationMode of a Port.

Valid values are defined in RsRegistrationMode enumeration.

Derived from RichType

RsGenericity

Description

Enumeration used to set the Value property of the Genericity Rich Type.

Public Attributes

rsFixed : Integer = 1

Description

A capsule of the specified class is automatically instantiated into the role in every instance of the container capsule at run-time. A number of instances equal to the specified cardinality will be created at initialization time.

rsOptional : Integer = 2

Description

The capsule role is instantiated under the program control of the container class. The container class must explicitly instantiate the capsule role within the detailed code of the container capsule state machine.

rsPlugIn : Integer = 3

Description

The capsule role is never directly instantiated, but rather an already existing instantiation from another capsule decomposition is imported into the role. That is, an existing capsule is dynamically “plugged in” to the specified role under the program control of the container class. The container class state machine must explicitly request the plug-in of a capsule at run-time within the detailed code.

RsRegistrationMode

Description

Enumeration used to set the Value property of the Registration Rich Type.

Public Attributes

rsApplication : Integer = 2

Description

The connection of non-wired ports is not connected at initialization time, it is connected when the capsule's behavior invokes a service function to register the port by a specified name. The same port may in fact be registered under different names at different points in the model execution.

rsAutomatic : Integer = 1

Description

The connection of non-wired ports is done automatically by name at the time the capsule is initialized.

rsNoMode : Integer = 0

Description

No registration mode specified.

Common Logical View Enumerations

Common Logical View Enumerations include

- *RsContainment* on page 372
    - Public Attributes

      rsByVal : Integer = 1 on page 372

      rsRef : Integer = 2 on page 373

      rsUnspecified : Integer = 0 on page 373
- *RsVisibilityKind* on page 373
    - Public Attributes

      rsImplementation : Integer = 3 on page 373

      rsPrivate : Integer = 2 on page 373

      rsProtected : Integer = 1 on page 374

      rsPublic : Integer = 0 on page 374

RsContainment

Description

Enumeration used to set the Value property of the AttributeContainment and the AssociationEndContainment Rich Types.

Public Attributes

rsByVal : Integer = 1

Description

Containment by value.

rsRef : Integer = 2

Description

Containment by reference.

rsUnspecified : Integer = 0

Description

Containment undefined.

RsVisibilityKind

Description

Enumeration used to set the Value property of the following Rich Types:

- *ClassifierVisibilityKind* on page 327
- *AttributeVisibilityKind* on page 340
- *OperationVisibilityKind* on page 345
- *AssociationEndVisibilityKind* on page 299
- *PortVisibilityKind* on page 370
- *GeneralizationVisibilityKind* on page 438
- *UsesRelationVisibilityKind* on page 444

Public Attributes

rsImplementation : Integer = 3

Description

Accessible only to the classifier itself.

rsPrivate : Integer = 2

Description

Accessible only to the classifier itself or to its friends.

rsProtected : Integer = 1

Description

Accessible only to subclasses, friends, or to the classifier itself.

rsPublic : Integer = 0

Description

Accessible to all clients.

Interaction Classes

Interaction classes include

- *Environment* on page 376
- *Interaction* on page 376
    - Public Attributes

      Instances : InteractionInstanceCollection on page 377

      Messages : MessageCollection on page 377

      ParentCollaboration : Collaboration on page 377

      ParentProtocol : Protocol on page 377

      SequenceDiagram : SequenceDiagram on page 377
    - Public Operations

      AddInteractionInstance (name : String) : InteractionInstance on page 377

      AddMessage (name : String, sender : InteractionInstance, receiver : InteractionInstance) : Message on page 378

      AddMessageWithAction (name : String, sender : InteractionInstance, receiver : InteractionInstance, ActionKind : RsActionKind) : Message on page 379

      DeleteInteractionInstance (theInstance : InteractionInstance) : Boolean on page 380

      DeleteMessage (theMessage : Message) : Boolean on page 380

      GetOwnerClassifierContext () : Classifier on page 381

ReorderInteractionInstance (theInstance : InteractionInstance, pBefore : InteractionInstance) : Boolean on page 381

ReorderMessage (theMessage : Message, pInsertBefore : Message) : Boolean on page 382

- *InteractionInstance* on page 382
  - Public Attributes

    ClassifierRoles : ClassifierRoleCollection on page 383

    Events : MessageEndCollection on page 383

    ParentInteraction : Interaction on page 383

    RootClassifier : Classifier on page 383
  - Public Attributes

    AddClassifierRole (theRole : ClassifierRole) : Boolean on page 383

    RemoveClassifierRole (theRole : ClassifierRole) : Boolean on page 384

    ReorderMessageEnd (theEnd : MessageEnd, pBefore : MessageEnd) : Boolean on page 385
- *Message* on page 385
  - Public Attributes

    Action : Action on page 386

    Activator : Message on page 386

    ParentInteraction : Interaction on page 386

    ReceiverEnd : MessageEnd on page 386

    SenderEnd : MessageEnd on page 386
- *MessageEnd* on page 386
  - Public Attributes

    Instance : InteractionInstance on page 387

    ParentMessage : Message on page 387
- *RsActionKind* on page 387
  - Public Attributes

    rsCallAction : Integer = 1 on page 387

rsCoregion : Integer = 5 on page 387

rsCreateAction : Integer = 4 on page 387

rsDestroyAction : Integer = 3 on page 387

rsLocalState : Integer = 2 on page 387

rsSendAction : Integer = 8 on page 387

rsTerminateAction : Integer = 7 on page 387

rsUninterpretedAction : Integer = 6 on page 387

Environment

Description

An Environment is an Interaction Instance associated with a Sequence Diagram's Environment View. This latter consists of the rectangular perimeter around the Sequence Diagram. It represents the external environment (hardware timers, SAPs/SPPs...) which can be communicated with but are not contained in the Capsule hierarchy. While it is possible to Send or Receive Call or Send messages it cannot be used as the Receiver of a "Destroy" message or as the location for Local States or Actions.

Derived from InteractionInstance

Interaction

Description

An Interaction is a Model Element associated with a Sequence Diagram. It contains the various Model Elements involved in the communication patterns described in the Sequence Diagram.

Derived from ModelElement

Public Attributes

Instances : InteractionInstanceCollection

Description

Interaction instances involved into the communication pattern expressed by the Interaction.

Messages : MessageCollection

Description

Messages involved into the communication pattern expressed by the Interaction.

ParentCollaboration : Collaboration

Description

Collaboration owning the Interaction. May be nothing if the Interaction is owned by a Protocol.

ParentProtocol : Protocol

Description

Protocol owning the Interaction. May be nothing if the Interaction is owned by a Collaboration.

SequenceDiagram : SequenceDiagram

Description

Diagram showing the communication patterns described by the Interaction.

Public Operations

AddInteractionInstance (name : String) : InteractionInstance

Description

Adds a new InteractionInstance into the Interaction and returns it.

Syntax

```
Set theInteractionInstance = theInteraction.AddInteractionInstance(
name )
```

```
theInteractionInstance As RoseRT.InteractionInstance
```

Returns the new InteractionInstance added to the Interaction.

```
theInteraction As RoseRT.Interaction
```

Interaction to which the InteractionInstance is being added.

```
name As String
```

Name of the new Interaction Instance added to the Interaction.

AddMessage (name : String, sender : InteractionInstance, receiver : InteractionInstance) : Message

Description

Adds a new Message into the Interaction and returns it. The action of the message is a Send Action.

Syntax

```
Set theMessage = theInteraction.AddMessage( name, sender, receiver )
```

```
theMessage As RoseRT.Message
```

Returns the new Message added to the Interaction.

```
theInteraction As RoseRT.Interaction
```

Interaction to which the message is being added.

```
name As String
```

Name of the new message added to the Interaction.

```
sender As RoseRT.InteractionInstance
```

Interaction Instance that sends the newly created message.

```
receiver As RoseRT.InteractionInstance
```

Interaction Instance that received the newly created message.

AddMessageWithAction (name : String, sender : InteractionInstance, receiver : InteractionInstance, ActionKind : RsActionKind) : Message

Description

Creates a new Message with an action of type specified and adds it into the Interaction and returns it.

Syntax

```
Set theMessage = theInteraction.AddMessage( name, sender, receiver,
ActionKind )
```

```
theMessage As RoseRT.Message
```

Returns the new Message added to the Interaction.

```
theInteraction As RoseRT.Interaction
```

Interaction to which the message is being added.

```
name As String
```

Name of the new message added to the Interaction.

```
sender As RoseRT.InteractionInstance
```

Interaction Instance that sends the newly created message.

```
receiver As RoseRT.InteractionInstance
```

Interaction Instance that received the newly created message.

```
ActionKind  As RoseRT.RsActionKind
```

Kind of action to add to message.

Note: sender and receiver should be the same interaction instances when ActionKind is one of rsLocalState, rsCoregion or rsUninterpretedAction.

DeleteInteractionInstance (theInstance : InteractionInstance) : Boolean

Description

Deleted an InteractionInstance from the Interaction.

Syntax

```
Deleted = theInteraction.DeleteInteractionInstance( theInstance )
```

```
Deleted As Boolean
```

Returns a value of True when the InteractionInstance is being deleted successfully from the Interaction.

```
theInteraction As RoseRT.Interaction
```

Interaction from which the InteractionInstance is being deleted.

```
theInstance As RoseRT.InterationInstance
```

Interaction Instance to delete from the Interaction.

DeleteMessage (theMessage : Message) : Boolean

Description

Deleted a Message from the Interaction.

Syntax

```
Deleted = theInteraction.DeleteMessage( theMessage )
```

```
Deleted As Boolean
```

Returns a value of True when the message is being deleted successfully from the Interaction.

```
theInteraction As RoseRT.Interaction
```

Interaction from which the Message is being deleted.

```
theMessage As RoseRT.Message
```

Message to delete from the Interaction.

GetOwnerClassifierContext () : Classifier

Description:

Gets the owner context of the particular interaction. This is useful for interactions that are owned by the structure of derived capsules. There is no path to the derived capsule except through this API.

Syntax:

```
Set theClassifier = theInteraction.GetOwnerClassifierContext()
```

```
theClassifier As RoseRT.Classifier
```

Returns the classifier that owns the collaboration that owns the interaction

ReorderInteractionInstance (theInstance : InteractionInstance, pBefore : InteractionInstance) : Boolean

Description

Reorders an InteractionInstance within the Interaction.

Syntax

```
Reordered = theInteraction.ReorderInteractionInstance( theInstance,
pBefore )
```

```
Reordered As Boolean
```

Returns a value of True when the reordering gets executed successfully.

```
theInteraction As RoseRT.Interaction
```

Interaction whose InteractionInstance is being reordered.

```
theInstance As RoseRT.InterationInstance
```

The Interaction Instance to be reordered.

```
pBefore As RoseRT.InterationInstance
```

The Interaction Instance to precede theInstance.

ReorderMessage (theMessage : Message, pInsertBefore : Message) : Boolean

Description

Reorders a Message within the Interaction.

Syntax

```
Reordered = theInteraction.ReorderMessage( theMessage, pInsertBefore )
```

```
Reordered As Boolean
```

Returns a value of True when the reordering gets executed successfully.

```
theInteraction As RoseRT.Interaction
```

Interaction whose message is being reordered.

```
theMessage As RoseRT.Message
```

The Message Instance to be reordered.

```
pInsertBefore As RoseRT.Message
```

The Message to precede theInstance.

InteractionInstance

Description

Model Element that maps to the Interaction Instance View of a Sequence Diagram.

Derived from ModelElement

Public Attributes

ClassifierRoles : ClassifierRoleCollection

Description

Identifies an object role in a collaboration to which the interaction instance is mapped. This property's type is a Collection representing a path to the mapped Classifier Role. Each element in the Collection corresponds to an element of the path. The last element is the actual Classifier Role the Interaction Instance maps to. Use with extreme care.

Events : MessageEndCollection

Description

Message Ends involved in the communication pattern described by the Interaction.

ParentInteraction : Interaction

Description

Interaction owning the Interaction Instance.

RootClassifier : Classifier

Description

Classifier whose projection is the ClassifierRole this InteractionInstance represents.

Public Operations

AddClassifierRole (theRole : ClassifierRole) : Boolean

Description

Adds a Classifier Role at the end of the path leading to the Classifier Role mapped by the Interaction Instance.

Syntax

```
Added = theInteractionInstance.AddClassifierRole( theRole )


Added As Boolean
```

Returns a value of True if the Classifier Role is added successfully at the end of the path.

```
theInteractionInstance As RoseRT.InteractionInstance
```

Interaction Instance whose mapped Classifier Role path gets added a Classifier Role.

```
theRole As RoseRT.ClassifierRole
```

Classifier Role that gets added at the end of the path leading to the Classifier Role mapped by the Interaction Instance.

RemoveClassifierRole (theRole : ClassifierRole) : Boolean

Description

Removes a Classifier Role from the path leading to the Classifier Role mapped by the Interaction Instance.

Syntax

```
Deleted = theInteractionInstance.RemoveClassifierRole( theRole )
```

```
Deleted As Boolean
```

Returns a value of True if the Classifier Role is removed successfully from the path.

```
theInteractionInstance As RoseRT.InteractionInstance
```

Interaction Instance whose mapped Classifier Role path gets removed a Classifier Role.

```
theRole As RoseRT.ClassifierRole
```

Classifier Role that gets removed from the path leading to the Classifier Role mapped by the Interaction Instance.

ReorderMessageEnd (theEnd : MessageEnd, pBefore : MessageEnd) : Boolean

Description

Reorders a Message End within the Interaction Instance.

Syntax

```
Reordered = theInteractionInstance.ReorderMessageEnd( theEnd, pBefore
)
```

`Reordered As Boolean`

Returns a value of True when the reordering gets executed successfully.

`theInteractionInstance As RoseRT.InteractionInstance`

Interaction Instance whose message end is being reordered.

`theEnd As RoseRT.MessageEnd`

The Message End to be reordered.

`pBefore As RoseRT.MessageEnd`

The Message End to precede theEnd.

Message

Description

A message defines how a particular request is used in an Interaction.

Derived from ModelElement

Public Attributes

Action : Action

Description

Action executed upon message activation.

Activator : Message

Description

Message activating the message.

ParentInteraction : Interaction

Description

Interaction owning the message.

ReceiverEnd : MessageEnd

Description

Message End connecting to the Interaction Instance receiving the message.

SenderEnd : MessageEnd

Description

Message End connecting to the Interaction Instance sending the message.

MessageEnd

Description

Links a Message to an Interaction Instance.

Derived from ModelElement

Public Attributes

Instance : InteractionInstance

Description

Interaction Instance linked by the Message End.

ParentMessage : Message

Description

Message linked by the Message End.

RsActionKind

Public Attributes

rsCallAction : Integer = 1

rsCoregion : Integer = 5

rsCreateAction : Integer = 4

rsDestroyAction : Integer = 3

rsLocalState : Integer = 2

rsSendAction : Integer = 8

rsTerminateAction : Integer = 7

rsUninterpretedAction : Integer = 6

State Machine Classes

State Machine classes include

- *RsSourceRegionType* on page 388
    - Public Attributes

rsFalseSourceRegion : Integer = 0 on page 389

rsTrueSourceRegion : Integer = 1 on page 389

- *SourceRegionType* on page 389
- *StateMachine* on page 389
  - Public Attributes

    Diagram : StateDiagram on page 389

    ParentClassifier : Classifier on page 390

    Top : CompositeState on page 390

    GetAllStates () : StateVertexCollection on page 390
- Transition
  - Public Attributes

    Action : Action on page 391

    EventGuards : EventGuardCollection on page 391

    Internal : Boolean on page 391

    ParentState : CompositeState on page 391

    ParentStateMachine : StateMachine on page 391

    Source : StateVertex on page 391

    SourceRegion : SourceRegionType on page 392

    Target : StateVertex on page 392
  - Public Operations

    AddEventGuard () : EventGuard on page 392

    DeleteEventGuard (theEventGuard : EventGuard) : Boolean on page 392

    SetUninterpretedAction (action : String) : UninterpretedAction on page 393

RsSourceRegionType

Description

Enumeration used to set the Value property of the SourceRegionType Rich Type.

Public Attributes

rsFalseSourceRegion : Integer = 0

Description

Source region associated to a FALSE transition.

rsTrueSourceRegion : Integer = 1

Description

Source region associated to a TRUE transition.

SourceRegionType

Description

Rich type used to determine SourceRegion property of a Transition. Also used when adding a transition to a Choice Point.

Valid values are defined in RsSourceRegionType enumeration.

Derived from RichType

StateMachine

Description

Class responsible for specifying the behavior on a Classifier.

Derived from Element

Public Attributes

Diagram : StateDiagram

Description

State Diagram projection of the State Machine.

ParentClassifier : Classifier

Description

Classifier owning the State Machine.

Top : CompositeState

Description

Composite State at the top of the State Machine.

Public Operations

GetAllStates () : StateVertexCollection

Description

Returns all states owned by the State Machine.

Syntax

```
Set theStateVertexCollection = theStateMachine.GetAllStates()
```

```
theStateVertexCollection As RoseRT.StateVertexCollection
```

Returns the collection of all states owned by the State Machine.

```
theStateMachine As RoseRT.StateMachine
```

The State Machine to retrieve owned states from.

Transition

Description

A transition is a relationship between two states, a source state and a destination state. It specifies that when an object in the source state receives a specified event and certain conditions are meet, the behavior will move from the source state to the destination state.

Derived from ModelElement

Public Attributes

Action : Action

Description

Action executed when a transition is triggered. For capsules, the transition action code will be output as part of the generated code, and the code will be executed when the transition is triggered at run-time. Transition actions defined in state diagrams for protocols or regular (non-capsule) classes is not generated or executed, it is for information purposes only.

EventGuards : EventGuardCollection

Description

Collection of Event Guards used to determine whether the transition should be triggered.

Internal : Boolean

Description

Indicates that a self-transition should not cause an exit from the state when triggered. The result is that when an internal transition is triggered, no exit or entry code is run.

ParentState : CompositeState

Description

Composite State owning the transition.

ParentStateMachine : StateMachine

Description

State Machine owning the parent state.

Source : StateVertex

Description

State at source end of the transition.

SourceRegion : SourceRegionType

Description

When the source of the transition is a Choice Point, determines whether the transition occurs on a TRUE or FALSE evaluation of the Choice Point condition. Irrelevant for other type of source state.

Target : StateVertex

Description

State at target end of the transition.

Public Operations

AddEventGuard () : EventGuard

Description

Adds a new event guard to the Transition.

Syntax

```
Set theEventGuard = theTransition.AddEventGuard()
```

```
theEventGuard As RoseRT.EventGuard
```

Returns the Event Guard added to the Transition.

```
theTransition As RoseRT.Transition
```

Transition to which a new event guard is being added.

DeleteEventGuard (theEventGuard : EventGuard) : Boolean

Description

Deletes an event guard from the Transition.

Syntax

```
Deleted = theTransition.DeleteEventGuard( theEventGuard )
```

```
Deleted As Boolean
```

Returns a value of True when the Event Guard is deleted successfully from the Transition.

```
theTransition As RoseRT.Transition
```

Transition from which an event guard is being deleted.

```
theEventGuard As RoseRT.EventGuard
```

The Event Guard deleted from the Transition.

SetUninterpretedAction (action : String) : UninterpretedAction

Description

Sets the action to execute when the transition is triggered.

Syntax

```
Set theUninterpretedAction = theTransition.SetUninterpretedAction(
action )
```

```
theUninterpretedAction As RoseRT.UninterpretedAction
```

Returns the new Uninterpreted Action to execute when the transition is triggered.

```
theTransition As RoseRT.Transition
```

Transition to which an uninterpreted action is being set.

```
action As String
```

The body of the new uninterpreted action.

Action Classes

Action Classes include

- *Action* on page 396
    - Public Attributes

      Arguments : StringCollection on page 396

      ParentMessage : Message on page 396

ParentState : CompositeState on page 396

ParentTransition : Transition on page 396

Time : String on page 397

- Public Operations

Action () : Action on page 397

AddArgument (szArg : String, nPosition : Integer) : Boolean on page 397

DeleteArgument (nPosition : Integer) : Boolean on page 398

- *ActionMode* on page 398
- *CallAction* on page 399
  - Public Attributes

  Operation : String on page 399
- *Coregion* on page 399
  - Public Attributes

  Events : MessageEndCollection on page 399
  - Public Operations

  AddEvent (event : MessageEnd) : Boolean on page 399

  RemoveEvent (event : MessageEnd) : Boolean on page 400

  ReorderEvent (event : MessageEnd, pBefore : MessageEnd) : Boolean on page 400
- *CreateAction* on page 401
  - Public Attributes

  Operation : String on page 401
- *DestroyAction* on page 401
- *LocalState* on page 402
- *ReplyAction* on page 402
  - Public Attributes

  Signal : String on page 402
- *RequestAction* on page 402
  - Public Attributes

Mode : ActionMode on page 403

Return : ResponseAction on page 403

- Public Operations

RequestAction () : RequestAction on page 403

- *ResponseAction* on page 403
  - Public Attributes

Request : RequestAction on page 404

- *ReturnAction* on page 404
- *RsActionMode* on page 404
  - Public Operations

rsAsynchronousMode : Integer = 1 on page 404

rsSynchronousMode : Integer = 0 on page 404

- *RsSendActionPriority* on page 405
  - Public Attributes

rsBackground : Integer = 5 on page 405

rsGeneral : Integer = 3 on page 405

rsHigh : Integer = 2 on page 405

rsLow : Integer = 4 on page 405

rsPanic : Integer = 1 on page 405

rsSystem : Integer = 0 on page 406

- *SendAction* on page 406
  - Public Attributes

DeliveryTime : String on page 406

Priority : SendActionPriority on page 406

ReceiverPort : String on page 406

SenderPort : String on page 406

Signal : String on page 406

- *SendActionPriority* on page 407

- *TerminateAction* on page 407
- *UninterpretedAction* on page 407
    - Public Attributes

      Body : String on page 407

Action

Description

Actions are the things the behavior does when a transition is taken. They represent executable atomic computations that are written as statements in a detail-level programming language and incorporated into a state machine. Actions are atomic, in the sense that they cannot be interrupted by the arrival of a higher priority event. An action therefore runs to completion.

Derived from ModelElement

Public Attributes

Arguments : StringCollection

Description

Name of arguments passed to the action.

ParentMessage : Message

Description

Message owning the Action. Nothing if the Action is owned by a State or a Transition.

ParentState : CompositeState

Description

State owning the Action. Nothing if the Action is owned by a Message or a Transition.

ParentTransition : Transition

Description

Transition owning the Action. Nothing if the Action is owned by a Message or a State.

Time : String

Description

Capture the time of the state change.

Public Operations

Action () : Action

Description

Returns an Action derived class as an Action.

Syntax

```
theCastedAction = theAction.Action()
```

```
theCastedAction As RoseRT.Action
```

Returns the Action derived class as an Action.

```
theAction As RoseRT.Action
```

Action to cast to an Action.

AddArgument (szArg : String, nPosition : Integer) : Boolean

Description

Adds an argument to the argument list of the action.

Syntax

```
Added = theAction.AddArgument( szArg, nPosition )
```

```
Added As Boolean
```

Returns a value of True when the argument is added successfully to the action's arguments' list.

```
theAction As RoseRT.Action
```

Action to which an argument is being added.

```
szArg As String
```

Name of the argument added to the action arguments' list.

```
nPosition As Integer
```

Position of the new argument in the action argument list.

DeleteArgument (nPosition : Integer) : Boolean

Description

Deletes an argument from the argument list of the action.

Syntax

```
Deleted = theAction.DeleteArgument( nPosition )
```

```
Deleted As Boolean
```

Returns a value of True when the argument is deleted successfully from the action's arguments' list.

```
theAction As RoseRT.Action
```

Action to which an argument is being deleted.

```
nPosition As Integer
```

Position of the argument to deleted from the action argument list.

ActionMode

Description

Rich type used to determine the Mode of a RequestAction.

Valid values are defined in RsActionMode enumeration.

Derived from RichType

CallAction

Description

Action resulting in the synchronous invocation of an operation on an instance.

Derived from RequestAction

Public Attributes

Operation : String

Description

Name of the receiver operation to call upon execution of the action.

Coregion

Description

Identifies a collection of incoming and outgoing messages where the order in which these messages are received/sent is not important.

Derived from Action

Public Attributes

Events : MessageEndCollection

Description

Message Ends connecting to messages that belong to the coregion.

Public Operations

AddEvent (event : MessageEnd) : Boolean

Description

Adds a Message End within the coregion.

Syntax

```
Added = theCoregion.AddEvent( event )
```

```
Added As Boolean
```

Returns a value of True when the Message End is added successfully to the coregion.

```
theCoregion As RoseRT.Coregion
```

Coregion to which a Message End is being added.

```
event As RoseRT.MessageEnd
```

Message End to add within the coregion.

RemoveEvent (event : MessageEnd) : Boolean

Description

Removes a Message End from within the coregion.

Syntax

```
Removed = theCoregion.RemoveEvent( event )
```

```
Removed As Boolean
```

Returns a value of True when the Message End is removed successfully from the coregion.

```
theCoregion As RoseRT.Coregion
```

Coregion to which a Message End is being removed.

```
event As RoseRT.MessageEnd
```

Message End to remove from within the coregion.

ReorderEvent (event : MessageEnd, pBefore : MessageEnd) : Boolean

Description

Reorders a Message End within the coregion.

Syntax

```
Reordered = theCoregion.ReorderEvent( event, pBefore )
```

`Reordered As Boolean`

Returns a value of True when the reordering gets executed successfully.

`theCoregion As RoseRT.Coregion`

Coregion whose message end is being reordered.

`event As RoseRT.MessageEnd`

The Message End to be reordered.

`pBefore As RoseRT.MessageEnd`

The Message End to precede event.

CreateAction

Description

Action resulting in the creation of an instance of some classifier.

Derived from Action

Public Attributes

Operation : String

Description

Name of the receiver operation to call upon creation of the instance.

DestroyAction

Description

Action that results in the destruction of an object specified in the action.

Derived from Action

LocalState

Description

Specifies a local state of the instance it is attached to. May correspond to a state within the state machine of the class of that instance.

Derived from Action

ReplyAction

Description

Response action from a Send Message.

Derived from ResponseAction

Public Attributes

Signal : String

Description

The name of the signal from the ports' protocol.

RequestAction

Description

Action enforcing an answer from the receiving end.

Derived from Action

Public Attributes

Mode : ActionMode

Description

The Mode property is a RichType that specifies whether an action is synchronous.

Return : ResponseAction

Description

The Response Action of the Request Action.

Public Operations

RequestAction () : RequestAction

Description

Returns a RequestAction derived class as a RequestAction.

Syntax

```
theCastedRequestAction = theRequestAction.RequestAction()
```

```
theCastedRequestAction As RoseRT.RequestAction
```

Returns the RequestAction derived class as a RequestAction.

```
theRequestAction As RoseRT.RequestAction
```

RequestAction to cast to a RequestAction.

ResponseAction

Description

Action triggered as a response to a Request Action.

Derived from Action

Public Attributes

Request : RequestAction

Description

Request Action that triggers the Response Action.

ReturnAction

Description

Response action from a Call Message.

Derived from ResponseAction

RsActionMode

Description

Enumeration used to set the Value property of the ActionMode Rich Type.

Public Attributes

rsAsynchronousMode : Integer = 1

Description

Asynchronous action.

rsSynchronousMode : Integer = 0

Description

Synchronous action.

RsSendActionPriority

Description

Enumeration used to set the Value property of the SendActionPriority Rich Type.

Public Attributes

rsBackground : Integer = 5

Description

Lowest priority used for background-type activities.

rsGeneral : Integer = 3

Description

Used for most processing; also the default.

rsHigh : Integer = 2

Description

Used for high-priority processing.

rsLow : Integer = 4

Description

Used for low-priority processing.

rsPanic : Integer = 1

Description

rsSystem : Integer = 0

Description

SendAction

Description

Action that results in the sending of a Signal, synchronous or asynchronous.

Derived from RequestAction

Public Attributes

DeliveryTime : String

Description

The time the message was delivered.

Priority : SendActionPriority

Description

The priority at which the message is sent.

ReceiverPort : String

Description

The name of the port on the receiver capsule.

SenderPort : String

Description

The name of the port on the sender capsule.

Signal : String

Description

The name of the signal from the ports' protocol.

SendActionPriority

Description

Rich type used to determine the Priority of a SendAction.

Valid values are defined in RsSendActionPriority enumeration.

Derived from RichType

TerminateAction

Description

Action resulting in the self destruction of an instance.

Derived from Action

UninterpretedAction

Description

Action whose result is not classified.

Derived from Action

Public Attributes

Body : String

Description

Code describing the result of the Uninterpreted Action.

Event Classes

Event classes include

- *Event* on page 409
    - Public Attributes

ParentEventGuard : EventGuard on page 409

- *EventGuard* on page 409
  - Public Attributes

    Event : Event on page 409

    Guard : String on page 409

    ParentTransition : Transition on page 410
  - Public Operations

    CreateEvent (name : String) : Event on page 410

    CreatePortEvent () : PortEvent on page 410

    CreateProtocolRoleEvent () : ProtocolRoleEvent on page 411
- *PortEvent* on page 411
  - Public Attributes

    Ports : PortCollection on page 411

    Signals : SignalCollection on page 412
  - Public Operations

    AddPort (port : Port) : Boolean on page 412

    AddPortByName (pszPortName : String) : Boolean on page 412

    AddSignal (signal : Signal) : Boolean on page 413

    AddSignalByName (pszSignalName : String) : Boolean on page 413

    RemovePort (port : Port) : Boolean on page 414

    RemoveSignal (signal : Signal) : Boolean on page 414
- *ProtocolRoleEvent* on page 415
  - Public Attributes

    Signals : SignalCollection on page 412
  - Public Operations

    AddPort (port : Port) : Boolean on page 412

    AddPortByName (pszPortName : String) : Boolean on page 412

    AddSignal (signal : Signal) : Boolean on page 413

AddSignalByName (pszSignalName : String) : Boolean on page 413

RemovePort (port : Port) : Boolean on page 414

RemoveSignal (signal : Signal) : Boolean on page 414

Event

Description

Events trigger transitions.

Derived from ModelElement

Public Attributes

ParentEventGuard : EventGuard

Description

Event Guard owning the event.

EventGuard

Description

An EventGuard is a grouping of an Event and a Guard that will trigger a transition.

Derived from ModelElement

Public Attributes

Event : Event

Description

Event to be activated by Event Guard.

Guard : String

Description

Code guarding the Event.

ParentTransition : Transition

Description

Transition owning the Event Guard.

Public Operations

CreateEvent (name : String) : Event

Description

Created the Event to guard. Use only for events created for analysis. For code generation, use CreatePortEvent() and CreateProtocolRoleEvent().

Syntax

```
Set theEvent = theEventGuard.CreateEvent( name )
```

`theEvent As RoseRT.Event`

Returns the newly created event.

`theEventGuard As RoseRT.EventGuard`

Event Guard to which an event is being created.

`name As String`

Name of the new event to guard.

CreatePortEvent () : PortEvent

Description

Created the a Port Event to guard.

Syntax

```
Set theEvent = theEventGuard.CreatePortEvent()
```

`theEvent As RoseRT.PortEvent`

Returns the newly created Port Event.

```
theEventGuard As RoseRT.EventGuard
```

Event Guard to which a Port Event is being created.

CreateProtocolRoleEvent () : ProtocolRoleEvent

Description

Created the a Protocol Role Event to guard.

Syntax

```
Set theEvent = theEventGuard.CreateProtocolRoleEvent()
```

```
theEvent As RoseRT.ProtocolRoleEvent
```

Returns the newly created Protocol Role Event.

```
theEventGuard As RoseRT.EventGuard
```

Event Guard to which a Protocol Role Event is being created.

PortEvent

Description

Event that results from the reception of a Signal from a specified set of Signals on any Port from a specified set of Ports.

Derived from Event

Public Attributes

Ports : PortCollection

Descriptions:

Collection of ports whose signals trigger transitions.

Signals : SignalCollection

Descriptions:

Collection of signals that trigger transitions.

Public Operations

AddPort (port : Port) : Boolean

Description

Adds a Port to the collection of ports whose signals cause the event to trigger a transition.

Syntax

```
Added = thePortEvent.AddPort( port )
```

```
Added As Boolean
```

Returns a value of True when the port is added successfully to the Port Event.

```
thePortEvent As RoseRT.PortEvent
```

Port Event to which a port is being added.

```
port As RoseRT.Port
```

Port to add to the Port Event.

AddPortByName (pszPortName : String) : Boolean

Description

Adds a Port to the collection of ports whose signals cause the event to trigger a transition.

Syntax

```
Added = thePortEvent.AddPortByName( pszPortName )
```

```
Added As Boolean
```

Returns a value of True when the port is added successfully to the Port Event.

```
thePortEvent As RoseRT.PortEvent
```
Port Event to which a port is being added.

```
pszPortNameAs String
```
Fully qualified name of the port to add to the Port Event.

AddSignal (signal : Signal) : Boolean

Description

Adds a Signal to the collection of signals that cause the event to trigger a transition.

Syntax

```
Added = thePortEvent.AddSignal( signal )
```

```
Added As Boolean
```
Returns a value of True when the signal is added successfully to the Port Event.

```
thePortEvent As RoseRT.PortEvent
```
Port Event to which a signal is being added.

```
signal As RoseRT.Signal
```
Signal to add to the Port Event.

AddSignalByName (pszSignalName : String) : Boolean

Description

Adds a Signal to the collection of signals that cause the event to trigger a transition.

Syntax

```
Added = thePortEvent.AddSignalByName( pszSignalName )
```

```
Added As Boolean
```
Returns a value of True when the signal is added successfully to the Port Event.

```
thePortEvent As RoseRT.PortEvent
```

Port Event to which a signal is being added.

```
pszSignalName As String
```

Name of the signal to add to the Port Event.

RemovePort (port : Port) : Boolean

Description

Removes a Port from the collection of ports whose signals cause the event to trigger a transition.

Syntax

```
Removed = thePortEvent.RemovePort( port )
```

```
Removed As Boolean
```

Returns a value of True when the port is removed successfully from the Port Event.

```
thePortEvent As RoseRT.PortEvent
```

Port Event to which a port is being removed.

```
port As RoseRT.Port
```

Port to remove from the Port Event.

RemoveSignal (signal : Signal) : Boolean

Description

Removes a signal from the collection of signals that cause the event to trigger a transition.

Syntax

```
Removed = thePortEvent.RemoveSignal( signal )
```

```
Removed As Boolean
```

Returns a value of True when the signal is removed successfully from the Port Event.

```
thePortEvent As RoseRT.PortEvent
```

Port Event to which a signal is being removed.

```
signal As RoseRT.Signal
```

Signal to remove from the Port Event.

ProtocolRoleEvent

Description

Event that results from the reception of a Signal in a Protocol Role.

Derived from Event

Public Attributes

Signals : SignalCollection

Description

Collection of signals that trigger transitions.

Public Operations

AddSignal (signal : Signal) : Boolean

Description

Adds a Signal to the collection of signals that cause the event to trigger a transition.

Syntax

```
Added = theProtocolRoleEvent.AddSignal( signal )
```

```
Added As Boolean
```

Returns a value of True when the signal is added successfully to the Protocol Role Event.

```
theProtocolRoleEvent As RoseRT.ProtocolRoleEvent
```

Protocol Role Event to which a signal is being added.

```
signal As RoseRT.Signal
```

Signal to add to the Protocol Role Event.

RemoveSignal (signal : Signal) : Boolean

Description

Removes a signal from the collection of signals that cause the event to trigger a transition.

Syntax

```
Removed = theProtocolRoleEvent.RemoveSignal( signal )
```

```
Removed As Boolean
```

Returns a value of True when the signal is removed successfully from the Protocol Role Event.

```
theProtocolRoleEvent As RoseRT.ProtocolRoleEvent
```

Protocol Role Event to which a signal is being removed.

```
signal As RoseRT.Signal
```

Signal to remove from the Protocol Role Event.

State Classes

State classes include

- *ChoicePoint* on page 418
    - Public Attributes

      Condition : String on page 419

      FALSETransition : Transition on page 419

      InTransition : Transition on page 419

      TRUETransition : Transition on page 419

- *CompositeState* on page 419
    - Public Attributes

      EntryAction : Action on page 419

      ExitAction : Action on page 420

      States : StateVertexCollection on page 420

      Transitions : TransitionCollection on page 420
    - Public Operations

      AddState (type : RsStateKind) : StateVertex on page 420

      AddTransition (source : String, sourceRegion : RsSourceRegionType, target : String) : Transition on page 421

      AddTransitionUsingStates (source : StateVertex, sourceRegion : RsSourceRegionType, target : StateVertex) : Transition on page 421

      DeleteState (theState : StateVertex) : Boolean on page 422

      DeleteTransition (theTransition : Transition) : Boolean on page 423

      SetUninterpretedEntryAction (action : String) : UninterpretedAction on page 423

      SetUninterpretedExitAction (action : String) : UninterpretedAction on page 424
- *FinalState* on page 424
- *InitialPoint* on page 425
- *JunctionContinuationMode* on page 425
- *JunctionPoint* on page 425
    - Public Attributes

      Continuation : JunctionContinuationMode on page 425

      ExternallyVisible : Boolean on page 426
    - Public Operations

      IsEntry () : Boolean on page 426

      IsExit () : Boolean on page 426
- *RsJunctionContinuationMode* on page 427
    - Public Attributes

      rsDeepHistory : Integer = 2 on page 427

rsDefault : Integer = 0 on page 427

rsShallowHistory : Integer = 1 on page 427

rsTransition : Integer = 3 on page 427

- *RsStateKind* on page 427
  - Public Attributes

    rsChoicePoint : Integer = 4 on page 428

    rsFinalState : Integer = 2 on page 428

    rsInitialPoint : Integer = 1 on page 428

    rsJunctionPoint : Integer = 3 on page 428

    rsNormalState : Integer = 0 on page 428
- *StateKind* on page 428
- *StateVertex* on page 429
  - Public Attributes

    ParentCompositeState : CompositeState on page 429

    ParentStateMachine : StateMachine on page 429

    GetIncomingTransitions () : TransitionCollection on page 429

    GetOutgoingTransitions () : TransitionCollection on page 430

    GetStateVertex () : StateVertex on page 430

ChoicePoint

Description

Choice points allow a single transition to be split into two outgoing transition segments, each of which can terminate on a different state.

Derived from StateVertex

Public Attributes

Condition : String

Description

Condition to be evaluated in order to determine which of the TRUE or FALSE transition to trigger.

FALSETransition : Transition

Description

The transition to trigger if the condition is evaluated to FALSE.

InTransition : Transition

Description

The transition that cause the condition to be evaluated.

TRUETransition : Transition

Description

The transition to trigger if the condition is evaluated to TRUE.

CompositeState

Description

State which owns a set of substates.

Derived from StateVertex

Public Attributes

EntryAction : Action

Description

Action executed on entering the state.

ExitAction : Action

Description

Action executed on exiting the state.

States : StateVertexCollection

Description

Substates owned by the Composite State.

Transitions : TransitionCollection

Description

Transitions owned by the Composite State. These are the transitions connecting substates.

Public Operations

AddState (type : RsStateKind) : StateVertex

Description

Adds a substate to the Composite State.

Syntax

```
Set theStateVertex = theCompositeState.AddState( type )
```

```
theStateVertex As RoseRT.StateVertex
```

Returns the State Vertex added to the Composite State.

```
theCompositeState As RoseRT.CompositeState
```

Composite State to which a substate is being added.

```
type As RoseRT.RsRichType
```

Type of the substate to add to the Composite State.

AddTransition (source : String, sourceRegion : RsSourceRegionType, target : String) : Transition

Description

Adds a transition to the Composite State.

Syntax

```
Set theTransition = theCompositeState.AddTransition( source,
sourceRegion, target )
```

`theTransition As RoseRT.Transition`

Returns the transition added to the Composite State.

`theCompositeState As RoseRT.CompositeState`

Composite State to which a transition is being added.

`source As String`

Name of substate attached to the source end of the new transition.

`sourceRegion As RoseRT.RsSourceRegionType`

If the source state kind is ChoicePoint, determines which of the TRUE or FALSE evaluation of the condition should trigger the new transition.

For other source state kind, this parameter is ignored.

`target As String`

Name of substate attached to the target end of the new transition.

AddTransitionUsingStates (source : StateVertex, sourceRegion : RsSourceRegionType, target : StateVertex) : Transition

Description

Adds a transition to the Composite State.

Syntax

```
Set theTransition = theCompositeState.AddTransitionUsingStates(
source, sourceRegion, target )
```

`theTransition As RoseRT.Transition`

Returns the transition added to the Composite State.

`theCompositeState As RoseRT.CompositeState`

Composite State to which a transition is being added.

`source As RoseRT.StateVertex`

Substate attached to the source end of the new transition.

`sourceRegion As RoseRT.RsSourceRegionType`

If the source state kind is ChoicePoint, determines which of the TRUE or FALSE evaluation of the condition should trigger the new transition.

For other source state kind, this parameter is ignored.

`target As RoseRT.StateVertex`

Substate attached to the target end of the new transition.

DeleteState (theState : StateVertex) : Boolean

Description

Deletes a substate from the Composite State.

Syntax

```
Deleted = theCompositeState.DeleteState( theState )
```

`Deleted As Boolean`

Returns a Value of True if the substate is deleted successfully from the Composite State.

```
theCompositeState As RoseRT.CompositeState
```

Composite State from which a substate is being deleted.

```
theState As RoseRT.StateVertex
```

Substate to delete from the Composite State.

DeleteTransition (theTransition : Transition) : Boolean

Description

Deletes a transition from the Composite State.

Syntax

```
Deleted = theCompositeState.DeleteTransition( theTransition )
```

```
Deleted As Boolean
```

Returns a Value of True if the transition is deleted successfully from the Composite State.

```
theCompositeState As RoseRT.CompositeState
```

Composite State from which a transition is being deleted.

```
theTransition As RoseRT.Transition
```

Transition to delete from the Composite State.

SetUninterpretedEntryAction (action : String) : UninterpretedAction

Description

Sets the entry action to execute on entering the Composite State.

Syntax

```
Set theUninterpretedAction =
theCompositeState.SetUninterpretedEntryAction( action )
```

```
theUninterpretedAction As RoseRT.UninterpretedAction
```

Returns the new Uninterpreted Action to execute on entering the Composite State.

```
theCompositeState As RoseRT.CompositeState
```

Composite State to which an entry action is being set.

```
action As String
```

The body of the new uninterpreted entry action.

SetUninterpretedExitAction (action : String) : UninterpretedAction

Description

Sets the exit action to execute on exiting the Composite State.

Syntax

```
Set theUninterpretedAction =
theCompositeState.SetUninterpretedExitAction( action )
```

```
theUninterpretedAction As RoseRT.UninterpretedAction
```

Returns the new Uninterpreted Action to execute on exiting the Composite State.

```
theCompositeState As RoseRT.CompositeState
```

Composite State to which an exit action is being set.

```
action As String
```

The body of the new uninterpreted exit action.

FinalState

Description

The end state of a Composite State.

Derived from StateVertex

InitialPoint

Description

Initial state of a Composite State. The InitialPoint can only have one outgoing transition.

Derived from StateVertex

JunctionContinuationMode

Description

Rich type used to determine Continuation property of a JunctionPoint. Valid values are defined in RsJunctionContinuationMode enumeration.

Derived from RichType

JunctionPoint

Description

State that sits on the border of a Composite State whose main purpose is to allow the continuation and joining of transitions.

Derived from StateVertex

Public Attributes

Continuation : JunctionContinuationMode

Description

The Continuation property is a RichType that specifies the semantics for how the state history will be used when there is no continuing transition.

ExternallyVisible : Boolean

Description

Indicates whether the junction point is visible on the outside of the state boundary

Public Operations

IsEntry () : Boolean

Description

Indicates whether the junction point connects to an incoming transition.

Syntax

```
IsEntry = theJunctionPoint.IsEntry()
```

```
IsEntry As Boolean
```

Returns a value of True if the transition connected to the Junction Point is an incoming transition.

```
theJunctionPoint As RoseRT.JunctionPoint
```

Junction point used to evaluate IsEntry.

IsExit () : Boolean

Description

Indicates whether the junction point connects to an outgoing transition.

Syntax

```
IsExit = theJunctionPoint.IsExit()
```

```
IsExit As Boolean
```

Returns a value of True if the transition connected to the Junction Point is an outgoing transition.

```
theJunctionPoint As RoseRT.JunctionPoint
```

Junction point used to evaluate IsExit.

RsJunctionContinuationMode

Description

Enumeration used to set the Value property of the JunctionContinuationMode Rich Type.

Public Attributes

rsDeepHistory : Integer = 2

Description

Specifies that the state should return to deep history, meaning that all substates also return to history.

rsDefault : Integer = 0

Description

Specifies that the default (initial) transition should be run.

rsShallowHistory : Integer = 1

Description

Specifies that the junction state should return to shallow history.

rsTransition : Integer = 3

Description

The Transition continuation mode cannot be set, it is returned if there is an exiting/continuing transition from the junction point.

RsStateKind

Description

Enumeration used to set the Value property of the StateKind Rich Type.

Public Attributes

rsChoicePoint : Integer = 4

Description

Choice point.

rsFinalState : Integer = 2

Description

Final state.

rsInitialPoint : Integer = 1

Description

Initial state.

rsJunctionPoint : Integer = 3

Description

Junction point.

rsNormalState : Integer = 0

Description

Normal state.

StateKind

Description

Rich type used to determine the kind of state added to a Composite State. See CompositeState's AddState operation. Notice this rich type exists only to strengthen the duality between enum and rich type. It is not used in the RRTEI API.

Valid values are defined in RsStateKind enumeration.

Derived from RichType

StateVertex

Description

Abstract class base of all states that are the source and destination of transitions.

Derived from ModelElement

Public Attributes

ParentCompositeState : CompositeState

Description

Composite State owning the state. Nothing if the state is the top state of a state machine.

ParentStateMachine : StateMachine

Description

State Machine owning the topmost parent Composite State.

Public Operations

GetIncomingTransitions () : TransitionCollection

Description

Return the collection of all incoming transitions of the State Vertex.

Syntax

```
Set theTransitions = theStateVertex.GetIncomingTransitions()
```

```
theTransitions As RoseRT.TransitionCollection
```

The collection of all incoming transitions of the State Vertex.

```
theStateVertex As RoseRT.StateVertex
```

State vertex to return incoming transitions from.

GetOutgoingTransitions () : TransitionCollection

Description

Return the collection of all outgoing transitions of the State Vertex.

Syntax

```
Set theTransitions = theStateVertex.GetOutgoingTransitions()
```

`theTransitions As RoseRT.TransitionCollection`

The collection of all outgoing transitions of the State Vertex.

`theStateVertex As RoseRT.StateVertex`

State vertex to return outgoing transitions from.

GetStateVertex () : StateVertex

Description

Return a State Vertex derived class instance as a State Vertex.

Syntax

```
Set theCastedStateVertex = theStateVertex.GetStateVertex()
```

`theCastedStateVertex As RoseRT.StateVertex`

The State Vertex derived class instance casted as a State Vertex.

`theStateVertex As RoseRT.StateVertex`

State vertex derived class instance to cast as a State Vertex.

Relation Classes

Relation classes include

- *ClassDependency* on page 433
  - Public Attributes

    ClientCardinality : String on page 433

    InvolvesFriendship : Boolean on page 433

    SupplierCardinality : String on page 434

    Visibility : UsesRelationVisibilityKind on page 434
- *ClassRelation* on page 434
  - Public Operations

    GetContextClassifier () : Classifier on page 434

    GetSupplierClassifier () : Classifier on page 435
- *ComponentDependency* on page 435
  - Public Attributes

    ContextClass : Class on page 435

    ContextComponent : Component on page 436

    ContextComponentPackage : ComponentPackage on page 436

    SupplierClass : Class on page 436

    SupplierComponent : Component on page 436

    SupplierComponentPackage : ComponentPackage on page 436
- *Generalization* on page 436
  - Public Attributes

    FriendshipRequired : Boolean on page 437

    Virtual : Boolean on page 437

    Visibility : GeneralizationVisibilityKind on page 437
  - Public Operations

    GetContextPackage () : LogicalPackage on page 437

    GetSupplierPackage () : LogicalPackage on page 438

- *GeneralizationVisibilityKind* on page 438
- *InstantiateRelation* on page 438
  - Public Attributes

    ContextClass : Class on page 439

    SupplierClass : Class on page 439
- *LogicalPackageDependency* on page 439
  - Public Operations

    GetContextLogicalPackage () : LogicalPackage on page 439

    GetSupplierLogicalPackage () : LogicalPackage on page 440
- *RealizeRelation* on page 440
  - Public Operations

    GetContextCapsule () : Capsule on page 441

    GetContextClass () : Class on page 441

    GetContextComponent () : Component on page 441

    GetContextProtocol () : Protocol on page 441

    GetSupplierClass () : Class on page 441

    GetSupplierUseCase () : UseCase on page 441
- *Relation* on page 442
  - Public Attributes

    SupplierName : String on page 442
  - Public Operations

    GetClient () : ModelElement on page 442

    GetSupplier () : ModelElement on page 442

    HasClient () : Boolean on page 443

    HasSupplier () : Boolean on page 443
- *UsesRelationVisibilityKind* on page 444

ClassDependency

Description

The ClassDependency class exposes a set of attributes and operations that

- Determine the characteristics of dependencies between classes
- Allow you to retrieve class dependencies

Derived from ClassRelation

Public Attributes

ClientCardinality : String

Description

Specifies the number of clients allowable for the ClassDependency.

Syntax

```
ClassDependency.ClientCardinality
```

Property Type:

String

InvolvesFriendship : Boolean

Description

Indicates whether the ClassDependency involves friendship.

Syntax

```
ClassDependency.InvolvesFriendship
```

Property Type:

Boolean

SupplierCardinality : String

Description

Specifies the number of suppliers allowable for the ClassDependency.

Syntax

```
ClassDependency.SupplierCardinality
```

Property Type:

String

Visibility : UsesRelationVisibilityKind

Description

The Visibility property is a RichType that specifies how a class dependency is viewed outside of the owner class.

ClassRelation

Description

The ClassRelation class inherits from the Relation class and is the parent class of the ClassDependency, and InheritRelation classes.

Check the lists attributes and operations for details.

Derived from Relation

Public Operations

GetContextClassifier () : Classifier

Description

Retrieves the Classifier relation's context (client) classifier.

Syntax

```
Set theClassifier = theClassifierRelation.GetContextClassifier ()
```

```
theClassifier As RoseRT.Classifier
```

Returns the realize relation's context (client) classifier.

```
theClassifierRelation As RoseRT.ClassifierRelation
```

ClassifierRelation whose context classifier is being retrieved.

GetSupplierClassifier () : Classifier

Description

Retrieves the Classifier relation's supplier classifier.

Syntax

```
Set theClassifier = theClassifierRelation.GetSupplierClassifier ()
```

```
theClassifier As RoseRT.Classifier
```

Returns the realize relation's supplier classifier.

```
theClassifierRelation As RoseRT.ClassifierRelation
```

ClassifierRelation whose supplier classifier is being retrieved.

ComponentDependency

Description

Describes the context and supplier relationship between components, component packages and classes.

Derived from Relation

Public Attributes

ContextClass : Class

Description

Returns the client (owner) class of the dependency. Nothing if the owner is not a class.

ContextComponent : Component

Description

Returns the client (owner) component of the dependency. Nothing if the owner is not a component.

ContextComponentPackage : ComponentPackage

Description

Returns the client (owner) component package of the dependency. Nothing if the owner is not a component package .

SupplierClass : Class

Description

Returns the supplier class of the dependency. Nothing if the supplier is not a class.

SupplierComponent : Component

Description

Returns the supplier component of the dependency. Nothing if the supplier is not a component.

SupplierComponentPackage : ComponentPackage

Description

Returns the supplier component package of the dependency. Nothing if the supplier is not a component package.

Generalization

Description

Generalization indicates a hierarchical relationship between classifiers in which one classifier shares the structure and/or behavior of another classifier. The Generalization class exposes a set of attributes and operations that

- Determine the characteristics of Inherit Relations between classifiers
- Allow you to retrieve Inherit Relations

Check the lists of attributes and operations for complete information.

Derived from ClassRelation

Public Attributes

FriendshipRequired : Boolean

Description

Indicates whether the generalization requires friendship. Friendship can be required between a supplier and a client in the relationship.

Virtual : Boolean

Description

Indicates whether the generalization is virtual.

Visibility : GeneralizationVisibilityKind

Description

The Visibility property is a RichType that specifies how the client of a Generalization relation exposes the inherited features of the supplier.

Public Operations

GetContextPackage () : LogicalPackage

Description

Returns the context logical package. Nothing if the context is not a logical package.

Syntax

```
Set theLogicalPackage = theGeneralizationn.GetContextPackage()

theLogicalPackage As RoseRT.LogicalPackage
```

The logical package that is the context of the generalization.

```
theGeneralization As RoseRT.Generalization
```

The generalization to retrieve the context from.

GetSupplierPackage () : LogicalPackage

Description

Returns the supplier logical package. Nothing if the supplier is not a logical package.

Syntax

```
Set theLogicalPackage = theGeneralizationn.GetSupplierPackage()
```

```
theLogicalPackage As RoseRT.LogicalPackage
```

The logical package that is the supplier of the generalization.

```
theGeneralization As RoseRT.Generalization
```

The generalization to retrieve the supplier from.

GeneralizationVisibilityKind

Description

Rich type used to determine how a Generalization relation can be accessed from other Classifiers. Valid values are defined in RsVisibility enumeration.

Derived from RichType

InstantiateRelation

Description

Describes the instantiate relationship between a parametrized class and an instantiated class.

Derived from ClassRelation

Public Attributes

ContextClass : Class

Description

Context side of the instantiate relationship. The client is an instantiated class or an instantiated class utility.

SupplierClass : Class

Description

Supplier side of the instantiate relationship. The client is a parametrized class or an parametrized class utility.

LogicalPackageDependency

Description

The LogicalPackageDependency class allows you to define and manipulate dependency relationships between LogicalPackages.

See the list of attributes and operations for details.

Derived from Relation

Public Operations

GetContextLogicalPackage () : LogicalPackage

Description

Retrieves the context (client) LogicalPackage belonging to the given LogicalPackage dependency.

Syntax

```
Set theLogicalPackage =
theLogicalPackageDependency.GetContextLogicalPackage ( )
```

```
theLogicalPackage As RoseRT.LogicalPackage
```

Returns the context (client) LogicalPackage belonging to the LogicalPackage dependency.

```
theLogicalPackageDependency As RoseRT.LogicalPackageDependency
```

LogicalPackage dependency whose context LogicalPackage is being retrieved.

GetSupplierLogicalPackage () : LogicalPackage

Description

Retrieves the supplier LogicalPackage belonging to the given LogicalPackage dependency.

Syntax

```
Set theLogicalPackage =
theLogicalPackageDependency.GetSupplierLogicalPackage ( )
```

```
theLogicalPackage As RoseRT.LogicalPackage
```

Returns the supplier LogicalPackage belonging to the LogicalPackage dependency.

```
theLogicalPackageDependency As RoseRT.LogicalPackageDependency
```

LogicalPackage dependency whose supplier LogicalPackage is being retrieved.

RealizeRelation

Description

A realize relationship shows that the client realizes the operations defined by the supplier.

Derived from Relation

Public Operations

GetContextCapsule () : Capsule

Description

Context (Client) capsule of the realize relation. Nothing if the context is not a capsule.

GetContextClass () : Class

Description

Context (Client) class of the realize relation. Nothing if the context is not a class.

GetContextComponent () : Component

Description

Context (Client) component of the realize relation. Nothing if the context is not a component.

GetContextProtocol () : Protocol

Description

Context (Client) protocol of the realize relation. Nothing if the context is not a protocol.

GetSupplierClass () : Class

Description

Supplier class of the realize relation. Nothing if the supplier is not a class.

GetSupplierUseCase () : UseCase

Description

Supplier use case of the realize relation. Nothing if the supplier is not a use case.

Relation

Description

All relations (ClassRelation, Inherits, Has, Realizes) inherit from the Relation Class. Relation Class properties and methods allow you to specify and retrieve the client and supplier information for the relations in a model.

Check the lists of attributes and operations for details.

Derived from ModelElement

Public Attributes

SupplierName : String

Description

Specifies the name of the supplier belonging to the relation.

Public Operations

GetClient () : ModelElement

Description

Retrieves the ModelElement that is the client belonging to the Relation.

Syntax

```
theModelElement = theRelation.GetClient ( )
```

```
theModelElement As RoseRT.ModelElement
```

Returns the ModelElement that is the client belonging to the relation.

```
theRelation As RoseRT.Relation
```

Relation whose client is being retrieved.

GetSupplier () : ModelElement

Description

Retrieves the ModelElement that is the supplier belonging to the Relation.

Syntax

```
theModelElement = theRelation.GetSupplier ( )
```

```
theModelElement As RoseRT.ModelElement
```

Returns the ModelElement that is the supplier belonging to the relation.

```
theRelation As RoseRT.Relation
```

Relation whose supplier is being retrieved.

HasClient () : Boolean

Description

Indicates whether the relation has a client.

Syntax

```
HasClient = theRelation.HasClient ( )
```

```
HasClient As RoseRT.Relation
```

Returns a value of True if the relation has a client.

```
theRelation As RoseRT.Relation
```

Relation being checked for a client.

HasSupplier () : Boolean

Description

Indicates whether the relation has a supplier.

Syntax

```
HasSupplier = theRelation.HasSupplier ( )
```

```
HasSupplier As RoseRT.Relation
```

Returns a value of True if the relation has a supplier.

`theRelation As RoseRT.Relation`

Relation being checked for a supplier.

UsesRelationVisibilityKind

Description

Rich type used to determine how a Uses relation can be accessed from other Classifiers. Valid values are defined in RsVisibility enumeration.

Derived from RichType

Use Case View Classes

Use Case View classes include

- *UseCase* on page 445
    - Public Attributes

      ClassDiagrams : ClassDiagramCollection on page 445

      Rank : String on page 445
    - Public Operations

      AddAssociation (szSupplierAssociationEndName : String, szSupplierAssociationEndType : String) : Association on page 445

      AddClassDiagram (szName : String) : ClassDiagram on page 446

      AddGeneralization (szName : String, szParentName : String) : Generalization on page 446

      DeleteAssociation (pDispatchAssociation : Association) : Boolean on page 447

      DeleteClassDiagram (pIDispatch : ClassDiagram) : Boolean on page 447

      DeleteGeneralization (theGeneralization : Generalization) : Boolean on page 448

      GetAssociationEnds () : AssociationEndCollection on page 449

      GetAssociations () : AssociationCollection on page 449

      GetGeneralizations () : GeneralizationCollection on page 449

      GetSuperUseCases () : UseCaseCollection on page 450

UseCase

Description

The Use Case class exposes a set of properties and methods that allow you to define and manipulate the sets of class diagrams and scenario diagrams that comprise a model's use cases.

Check the lists of attributes and operations for complete information.

Derived from Classifier

Public Attributes

ClassDiagrams : ClassDiagramCollection

Description

Specifies the collection of class diagrams belonging to the use case

Rank : String

Description

Specifies the rank of the use case.

Public Operations

AddAssociation (szSupplierAssociationEndName : String, szSupplierAssociationEndType : String) : Association

Description

Adds an association to a use case and returns it in the specified object.

Syntax

```
Set theAssociation = theUseCase.AddAssociation (theSupplierRoleName,
theSupplierRoleType)
```

`theAssociation As RoseRT.Association`

Returns the association being added to the use case.

`theUseCase As RoseRT.UseCase`

Use case to which the association is being added.

`theSupplierRoleName As String`

Name of the supplier role in the association.

`theSupplierRoleType As String`

Type of the supplier role in the association.

AddClassDiagram (szName : String) : ClassDiagram

Description

Creates a new class diagram and adds it to a use case.

Syntax

```
Set theClassDiagram = theUseCase.AddClassDiagram (theName)
```

`theClassDiagram As RoseRT.ClassDiagram`

Returns the class diagram being added to the use case.

`theUseCase As RoseRT.UseCase`

UseCase to which the diagram is being added.

`theName As String`

The name of the class diagram to be added.

AddGeneralization (szName : String, szParentName : String) : Generalization

Description

This function adds a Generalization relationship to a use case and returns it in the specified object.

Syntax

```
Set theGeneralization = theUseCase.AddGeneralization( szName,
szParentName )
```

`theGeneralization As RoseRT.Generalization`

Returns the Generalization being added to the classifier.

`theUseCase As RoseRT.UseCase`

Use case to which the Generalization is being added.

`szName As String`

Name of the new Generalization.

`szParentName As String`

Name of the parent use case in the Generalize relationship.

DeleteAssociation (pDispatchAssociation : Association) : Boolean

Description

Deletes an association from a use case.

Syntax

```
Deleted = theUseCase.DeleteAssociation (theAssociation)
```

`Deleted As Boolean`

Returns a value of True when the association is deleted.

`theUseCase As RoseRT.UseCase`

Use case from which the association is being deleted.

`theAssociation As RoseRT.Association`

Instance of the association being deleted (The association must belong to the specified use case.)

DeleteClassDiagram (pIDispatch : ClassDiagram) : Boolean

Description

Deletes a class diagram from a use case.

Syntax

```
deleted = theUseCase.DeleteClassDiagram (theClassDiagram)
```

```
deleted As Boolean
```

Returns a value of True when the class diagram is deleted.

```
theUseCase As RoseRT.UseCase
```

Use case from which the class diagram is being deleted.

```
theClassDiagram As RoseRT.ClassDiagram
```

Instance of the class diagram being deleted.

DeleteGeneralization (theGeneralization : Generalization) : Boolean

Description

This function deleted a Generalization relation from a use case.

Syntax

```
Deleted = theUseCase.DeleteGeneralization( theGeneralization )
```

```
Deleted As Boolean
```

Returns a value of True when the generalization gets deleted successfully from the use case.

```
theUseCase As RoseRT.UseCase
```

Use case from which the generalization is being deleted.

```
theGeneralization As RoseRT.Generalization
```

The generalization being deleted.

GetAssociationEnds () : AssociationEndCollection

Description

Retrieves an AssociationEnd collection from a use case and returns it in the specified object.

Syntax

```
Set theAssociationEnds = theUseCase.GetAssociationEnds ( )
```

```
theAssociationEnds As RoseRT.AssociationEndCollection
```

Returns the AssociationEnd collection from the class.

```
theUseCase As RoseRT.UseCase
```

UseCase from which the collection is being retrieved.

GetAssociations () : AssociationCollection

Description

Retrieves an association collection from a use case and returns it in the specified object.

Syntax

```
Set theAssociations = theUseCase.GetAssociations
```

```
theAssociations As RoseRT.AssociationCollection
```

Returns the association collection from the use case.

```
theUseCase As RoseRT.UseCase
```

Use case from which the collection is being retrieved.

GetGeneralizations () : GeneralizationCollection

Description

Returns the set of Generalization a use case is client of.

Syntax

```
Set Generalizations = theUseCase.GetGeneralizations()
```

```
Generalizations As RoseRT.Classifier
```

The collection of all Generalization relationships the use case is client of.

```
theUseCase As RoseRT.UseCase
```

The use case to return Generalization it is client of.

GetSuperUseCases () : UseCaseCollection

Description

Retrieves a super use case collection from a use case and returns it in the specified object.

Syntax

```
Set theSuperUseCases = theUseCase.GetSuperUseCases ( )
theSuperUseCases As RoseRT.UseCaseCollection
```

Returns the super use case collection from the use case.

```
theUseCase As RoseRT.UseCase
```

Use case from which the collection is being retrieved.

View Classes

View classes include

- *AnchorNoteView* on page 453
    - Public Attributes

      Text : String on page 454
- *Diagram* on page 454
    - Public Attributes

      Documentation : String on page 454

      ExternalDocuments : ExternalDocumentCollection on page 454

ModelElements : ModelElementCollection on page 454

ParentModelElement : ModelElement on page 455

ViewElements : ViewElementCollection on page 455

Visible : Boolean on page 455

ZoomFactor : Integer on page 455

- Public Operations

Activate () : on page 455

AddAnchorNoteView (FromView : ViewElement, ToView : ViewElement) : AnchorNoteView on page 456

AddExternalDocument (szName : String, iType : RsExternalDocumentType) : ExternalDocument on page 456

AddNoteView (szNoteText : String, nType : RsNoteViewType) : NoteView on page 457

DeleteExternalDocument (theExtDoc : ExternalDocument) : Boolean on page 458

Exists (theModelElement : ModelElement) : Boolean on page 458

GetNoteViews () : NoteViewCollection on page 459

GetSelectedModelElements () : ModelElementCollection on page 459

GetViewFrom (theModelElement : ModelElement) : ViewElement on page 459

Invalidate () : on page 460

IsActive () : Boolean on page 460

Layout () : on page 461

RemoveAnchorNoteView (anchorNoteView : AnchorNoteView) : Boolean on page 461

RemoveNoteView (pIDispNoteView : NoteView) : Boolean on page 462

Render (FileName : String) : on page 462

RenderEnhanced (FileName : String) : on page 463

RenderEnhancedToClipboard () : on page 463

RenderToClipboard () : on page 463

Update () : on page 464

- *NoteView* on page 464

- Public Attributes

  Text : String on page 464

- Public Operations

  GetNoteViewType () : RsNoteViewType on page 465

  LinkToDiagram (diagramToLink : Diagram) : Boolean on page 465

- *RsNoteViewType* on page 466
  - Public Attributes

    rsConstraint : Integer = 3 on page 466

    rsFloatingTextLabel : Integer = 1 on page 466

    rsNoteWithBox : Integer = 2 on page 466
- *RsStereotypeDisplay* on page 466
  - Public Attributes

    rsDecorationAndLabel : Integer = 2 on page 467

    rsDecorationOnly : Integer = 3 on page 467

    rsIcon : Integer = 4 on page 467

    rsLabel : Integer = 1 on page 467

    rsNone : Integer = 0 on page 467
- *StereotypeDisplay* on page 467
- *ViewElement* on page 467
  - Public Attributes

    FillColor : View_FillColor on page 467

    Font : View_Font on page 468

    Height : Integer on page 468

    LineColor : View_LineColor on page 468

    LineVertices : LineVertexCollection on page 468

    ModelElement : ModelElement on page 468

    ParentDiagram : Diagram on page 468

    ParentView : ViewElement

StereotypeDisplay : StereotypeDisplay on page 469

SubViews : ViewElementCollection on page 469

Width : Integer on page 469

XPosition : Integer on page 469

YPosition : Integer on page 469

- Public Operations

GetDefaultHeight () : Integer on page 469

GetDefaultWidth () : Integer on page 470

GetMinHeight () : Integer on page 470

GetMinWidth () : Integer on page 471

HasModelElement () : Boolean on page 471

HasParentView () : Boolean on page 471

Invalidate () : on page 472

IsSelected () : Boolean on page 472

PointInView (x : Integer, y : Integer) : Boolean on page 473

SetSelected (bSelect : Boolean) : on page 473

SupportsFillColor () : Boolean on page 474

SupportsLineColor () : Boolean on page 474

AnchorNoteView

Description

The anchor note view class inherits the ViewElement attributes and operations that determine the size and placement of the anchor note view on a diagram.

Check the lists of attributes and operations for complete information.

Derived from ViewElement

Public Attributes

Text : String

Description

Contains the text that appears in the AnchorNoteView object.

Diagram

Description

The Diagram class exposes a set of attributes and operations, which all other diagram classes (for example, class diagrams, sequence diagrams, Collaboration diagrams, etc.) inherit. These attributes and operations determine the size and placement of a diagram on the Rose RealTime user's computer screen.

Check the lists of attributes and operations for complete information.

Derived from ControllableElement

Public Attributes

Documentation : String

Description

Specifies the documentation belonging to the Diagram.

ExternalDocuments : ExternalDocumentCollection

Description

Specifies the external documents belonging to the diagram.

ModelElements : ModelElementCollection

Description

Specifies the collection of ModelElements belonging to the diagram.

ParentModelElement : ModelElement

Description

Model element the diagram belongs to.

ViewElements : ViewElementCollection

Description

Specifies the collection of element views belonging to the diagram.

Visible : Boolean

Description

Indicates whether the diagram is visible on the computer sc

ZoomFactor : Integer

Public Operations

Activate () :

Description

Makes the specified diagram the active diagram in Rose RealTime. The active diagram is the window in Rose RealTime which currently has the focus.

Syntax

```
theDiagram.Activate
```

```
theDiagram As RoseRT.Diagram
```

Diagram to activate.

See also

IsActive Method

GetActiveDiagram Method

AddAnchorNoteView (FromView : ViewElement, ToView : ViewElement) : AnchorNoteView

Description:

Adds an anchor note view object to a diagram.

Syntax:

```
Set theAnchorNoteView = theDiagram.AddAnchorNoteView (theFromView,
theToView)
```

```
theAnchorNoteView as RoseRT.AnchorNoteView
```

Returns the anchor note view object added to the diagram.

```
theDiagram As RoseRT.Diagram
```

Diagram to which the anchor note view object is being added.

```
theFromView As RoseRT.ViewElement
```

ViewElement from which the note anchor starts at.

```
theToView As RoseRT.ViewElement
```

ViewElement to which the note anchor ends at.

AddExternalDocument (szName : String, iType : RsExternalDocumentType) : ExternalDocument

Description

Creates a new external document and adds it to a diagram.

Syntax

```
Added = theDiagram.AddExternalDocument (theName, theType)
```

```
Added As Boolean
```

Returns a value of true when the document is added to the diagram.

`theDiagram As RoseRT.Diagram`

Diagram to which the document is being added.

`theName As String`

Name of the document being added.

`theType As Integer`

Type of document being added Valid values are:

1 = Path

2 = URL

AddNoteView (szNoteText : String, nType : RsNoteViewType) : NoteView

Description

Adds a note view object to a diagram

Syntax

```
Set theNoteView = theDiagram.AddNoteView (theNoteText,
theNoteViewType)
```

`theNoteView as RoseRT.NoteView`

Returns the note view object added to the diagram.

`theDiagram As RoseRT.Diagram`

Diagram to which the note view object is being added.

`theNoteText As String`

Contains the text of the note view object.

`theNoteViewType As Integer`

Indicates whether the note is free floating or enclosed in a box:

1 = Free floating text label

2 = Note with box

DeleteExternalDocument (theExtDoc : ExternalDocument) : Boolean

Description

Deletes an external document from a diagram.

Syntax

```
Deleted = theDiagram.DeleteExternalDocument (theDocument)
```

`deleted As Boolean`

Returns a value of true when the document is deleted from the diagram.

`theDiagram As RoseRT.Diagram`

Diagram from which the document is being deleted.

`theDocument As RoseRT.ExternalDocument`

Instance of the document being deleted.

Exists (theModelElement : ModelElement) : Boolean

Description

Determines whether a specified diagram object exists.

Syntax

```
Exists = theDiagram.Exists (theModelElement)
```

`Exists As Boolean`

Returns the value of TRUE if the diagram object exists.

`theDiagram As RoseRT.Diagram`

Instance of the diagram whose existence is being checked.

```
theModelElement As RoseRT.ModelElement
```

Instance of the Rose item that corresponds to the diagram object.

GetNoteViews () : NoteViewCollection

Description

Returns the collection of note views belonging to a diagram.

Syntax

```
Set theNoteViews = theDiagram.GetNoteViews ( )
```

```
theNoteViews As RoseRT.NoteViewCollection
```

Returns the collection of note views belonging to the diagram.

```
theDiagram As RoseRT.Diagram
```

Instance of the diagram whose note view objects are being retrieved.

GetSelectedModelElements () : ModelElementCollection

Description

Returns all currently selected items in a diagram

Syntax

```
Set theItemCollection = theDiagram.GetSelectedItems ( )
```

```
theItemCollection As RoseRT.ItemCollection
```

Returns the Rose item view (view object) that represents the specified Rose item.

```
theDiagram As RoseRT.Diagram
```

Instance of the diagram whose selected items are being retrieved.

GetViewFrom (theModelElement : ModelElement) : ViewElement

Description

Retrieves the Rose item view that represents the specified Rose item.

Syntax

```
Set theView = theDiagram.GetViewFrom (theModelElement)
```

```
theView As RoseRT.ModelElementView
```

Returns the Rose item view (view object) that represents the specified Rose item.

```
theDiagram As RoseRT.Diagram
```

Instance of the diagram that contains the view object.

```
theModelElement As RoseRT.ModelElement
```

Instance of the Rose item whose view item is being returned.

Invalidate () :

Description

Invalidates a Rose diagram; that is, it causes the diagram to be redrawn.

Syntax

```
theDiagram.Invalidate
```

```
theDiagram As RoseRT.Diagram
```

Diagram being redrawn.

IsActive () : Boolean

Description

Indicates whether the diagram is the currently active diagram in the application

Syntax

```
IsActive = theDiagram.IsActive ()
```

```
IsActive As Boolean
```

Returns a value of True if the diagram is the current active in Rose; otherwise, returns a value of False.

```
theDiagram As RoseRT.Diagram
```
Diagram being checked as current diagram.

See also

Activate Method

GetActiveDiagram Method

Layout () :

Description

Draws a Rose RealTime diagram.

Syntax

```
theDiagram.Layout
```

```
theDiagram As RoseRT.Diagram
```
Diagram being drawn.

RemoveAnchorNoteView (anchorNoteView : AnchorNoteView) : Boolean

Description:

Removes an anchor note view object to a diagram

Syntax:

bRet = theDiagram.RemoveAnchorNoteView (theAnchorNoteView)

```
bRet as Boolean
```
True if the view was removed sucessfully, False otherwise.

```
theAnchorNoteView As RoseRT.AnchorNoteView
```
The anchor note view object which is being removed from the diagram.

RemoveNoteView (pIDispNoteView : NoteView) : Boolean

Description

Removes a note view object from a diagram

Syntax

```
Set IsRemoved = theDiagram.RemoveNoteView (theNoteView)
```

```
cIsRemoved As Boolean
```

Returns a value of True when the note view object is successfully removed.

```
theDiagram As RoseRT.Diagram
```

Diagram from which the note view object is being removed.

```
theNoteView as RoseRT.NoteView
```

Note view object to be removed from the diagram.

Render (FileName : String) :

Renders a Rose RealTime diagram to a Windows metafile, allowing the diagram to be opened and edited in any application that works with Windows metafiles.

Syntax

```
theDiagram.Render theFileName
```

```
theDiagram As RoseRT.Diagram
```

Diagram to render.

```
theFileName As String
```

Name of the Windows metafile in which to save the diagram.

RenderEnhanced (FileName : String) :

Description

Renders a Rose RealTime diagram to an enhanced Windows metafile, allowing the diagram to be opened and edited in any application that works with Windows metafiles.

Syntax

```
theDiagram.RenderEnhanced theFileName
```

```
theDiagram As RoseRT.Diagram
```

Diagram to render.

```
theFileName As String
```

Name of the enhanced Windows metafile in which to save the diagram.

RenderEnhancedToClipboard () :

Description

Renders a Rose RealTime diagram to the Clipboard, preserving its Enhanced metafile formatting information. As with any Clipboard object, it can then be pasted into other windows or compatible applications.

Syntax

```
theDiagram.RenderEnhancedToClipboard
```

```
theDiagram As RoseRT.Diagram
```

Diagram to render.

RenderToClipboard () :

Description

Renders a Rose RealTime diagram to the Clipboard in Windows metafile format. As with any Clipboard object, it can then be pasted into other windows or compatible applications.

Syntax

```
theDiagram.RenderToClipboard
```

```
theDiagram As RoseRT.Diagram
```

Diagram to render.

Update () :

Description

Updates a Rose RealTime diagram.

Syntax

```
theDiagram.Update
```

```
theDiagram As RoseRT.Diagram
```

Diagram being updated.

NoteView

Description

The note view class inherits the ModelElement attributes and operations that determine the size and placement of the note view on a diagram.

Check the lists of attributes and operations for complete information.

Derived from ViewElement

Public Attributes

Text : String

Description

Contains the text that appears in the NoteView object.

Public Operations

GetNoteViewType () : RsNoteViewType

Description

Returns the Type value of a NoteView object.

Syntax

```
theType = theNoteView.GetNoteViewType ()
```

```
theType As RsNoteViewType
```

Retrieves the integer value that corresponds to the NoteView type.

```
theNoteView As RoseRT.NoteView
```

Instance of the NoteView whose type is being retrieved.

LinkToDiagram (diagramToLink : Diagram) : Boolean

Description:

Allows a note to be linked to a specific diagram. When user double clicks on the note subsequently, the linked diagram will be opened up and activated.

Syntax:

```
theReturn = theNoteView.LinkToDiagram ( theDiagramToLink)
```

```
theReturn As Boolean
```

Returns whether the linkage was successful or not.

```
theNoteView As RoseRT.NoteView
```

Instance of the NoteView whose type is being retrieved.

```
theDiagramToLink As RoseRT.Diagram
```

Diagram the note will link to.

RsNoteViewType

Description

Enumeration used in NoteView::GetNoteViewType() and in Diagram::AddNoteView() to determine the type of the NoteView.

Public Attributes

rsConstraint : Integer = 3

Description

The Note View is a constraint

rsFloatingTextLabel : Integer = 1

Description

The Note View is floating text.

rsNoteWithBox : Integer = 2

Description

The Note View is a textual note with a box around it.

RsStereotypeDisplay

Description

Enumeration used to set the Value property of the StereotypeDisplay Rich Type.

Public Attributes

rsDecorationAndLabel : Integer = 2

rsDecorationOnly : Integer = 3

rsIcon : Integer = 4

rsLabel : Integer = 1

rsNone : Integer = 0

StereotypeDisplay

Description

Rich type used to how a view element stereotype will get displayed.

Valid values are defined in RsStereotypeDisplay enumeration.

Derived from RichType

ViewElement

Description

The ViewElement class exposes a set of attributes and operations that determine the size and placement of a ModelElement on a diagram.

Check the lists of attributes and operations for complete information.

Derived from Element

Public Attributes

FillColor : View_FillColor

Description

Specifies the amount of red, green, or blue to use in the fill color for the ModelElementView object, or whether it is transparent.

Font : View_Font

Description

Specifies the amount of red, green, or blue to use in the text color of a ModelElementView object.

Height : Integer

Description

Specifies the height of the object.

LineColor : View_LineColor

Description

Specifies the amount of red, green, or blue to use in the line color for the ModelElementView object.

LineVertices : LineVertexCollection

Description

Collection of line vertex objects representing the path of connector-like objects. Will be empty for non connector-like objects.

ModelElement : ModelElement

Description

Specifies the ModelElement represented by this ModelElementView.

ParentDiagram : Diagram

Description

Specifies the diagram that contains this ModelElementView.

ParentView : ViewElement

Description

Specifies the ModelElementView that contains this ModelElementView.

StereotypeDisplay : StereotypeDisplay

Description

The StereotypeDisplay property is a RichType that specifies how the stereotype of a model element will get displayed.

SubViews : ViewElementCollection

Description

Specifies the collection of item views that belong to the ModelElement.

Width : Integer

Description

Specifies the width of the ModelElement view.

XPosition : Integer

Description

Specifies the value of the horizontal coordinate (x) for the center point of the view.

YPosition : Integer

Description

Specifies the value of the vertical coordinate (y) for the center point of the view.

Public Operations

GetDefaultHeight () : Integer

Description

Retrieves the ideal height of the ModelElementView object, based on the object's formatting. This value is calculated by Rose and cannot be set.

Syntax

```
theHeight = theModelElementView.GetDefaultHeight ()


theHeight As RoseRT.Integer
```

Returns the ideal height of the ModelElementView, given the formatting of the object.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView whose ideal height you are determining.

GetDefaultWidth () : Integer

Description

Retrieves the ideal width of the ModelElementView object, based on the object's formatting. This value is calculated by Rose and cannot be set.

Syntax

```
theWidth = theModelElementView.GetDefaultWidth ()
```

```
theWidth As Integer
```

Returns the ideal width of the ModelElementView, given the formatting of the object.

```
theModelElementView As ModelElementView
```

Specifies the ModelElementView whose ideal width you are determining.

GetMinHeight () : Integer

Description

Retrieves the minimum height of the ModelElementView object, based on the object's formatting. This value is calculated by Rose and cannot be set.

Syntax

```
theHeight = theModelElementView.GetMinHeight ()
```

```
theHeight As Integer
```

Returns the minimum height of the ModelElementView, given the formatting of the object.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView whose minimum height you are determining.

GetMinWidth () : Integer

Description

Retrieves the minimum width of the ModelElementView object, based on the object's formatting. This value is calculated by Rose and cannot be set.

Syntax

```
theHeight = theModelElementView.GetMinWidth ()
```

```
theWidth As Integer
```

Returns the minimum width of the ModelElementView, given the formatting of the object.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView whose minimum width you are determining.

HasModelElement () : Boolean

Description

Indicates whether the ModelElementView has a corresponding ModelElement.

Syntax

```
HasItem = theModelElementView.HasItem ()
```

```
HasItem As Boolean
```

Returns a value of True if the ModelElementView has a corresponding ModelElement.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView being checked for a ModelElement.

HasParentView () : Boolean

Description

Indicates whether the ModelElementView belongs to another ModelElementView.

Syntax

```
HasParentView = theModelElementView.HasParentView ()
```

```
HasParentView As Boolean
```

Returns a value of True if the ModelElementView belongs to another ModelElementView.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView being checked for a parent view.

Invalidate () :

Description

Redraws the ModelElementView on the screen.

Syntax

```
theObject.Invalidate
```

```
theObject As RoseRT.ModelElementView
```

Instance of the ModelElementView being redrawn.

IsSelected () : Boolean

Description

Indicates whether the ModelElementView is currently selected in the diagram.

Syntax

```
IsSelected = theModelElementView.IsSelected ()
```

```
IsSelected As Boolean
```

Returns a value of True if the ModelElementView is currently selected in the diagram.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView being checked for in the diagram.

PointInView (x : Integer, y : Integer) : Boolean

Description

Determines whether a given x,y coordinate lies within the specified ModelElementView.

Syntax

```
InView = theModelElementView.PointInView ()
```

```
IsInView As Boolean
```

Returns a value of True if the given x,y coordinate lies within the specified ModelElementView.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView being checked for a ModelElement.

SetSelected (bSelect : Boolean) :

Description

Selects the given ModelElementView in the diagram.

Syntax

```
theModelElementView.SetSelected Selected
```

```
theModelElementView As RoseRT.ModelElementView
```

ModelElementView to select.

```
Selected As Boolean
```

Set to True to select the ModelElementView in the diagram; set to False to deselect the ModelElementView in the diagram.

SupportsFillColor () : Boolean

Description

Causes the ModelElementView to support fill color, if the type of ModelElementView can support fill color. For example, a ModelElementView that represents a class can use a fill color. However, a ModelElementView that represents a relationship line, it cannot support fill color. (It can, however, support a line color.)

Syntax

SupportsFill = theModelElementView.SupportsFillColor ()

```
SupportsFill As Boolean
```

Returns a value of True if the specified ModelElementView is to support a fill color.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView to support fill color.

SupportsLineColor () : Boolean

Description

Causes the ModelElementView to support line color, if the type of ModelElementView can support line color. For example, a ModelElementView that represents a relationship line can support line color. However, a ModelElementView that displays a metafile cannot support a line color.

Syntax

```
SupportsLine = theModelElementView.SupportsLineColor ()
```

```
SupportsLine As Boolean
```

Returns a value of True if the specified ModelElementView is to support a line color.

```
theModelElementView As RoseRT.ModelElementView
```

Specifies the ModelElementView to support line color.

Class Diagram Classes

Class Diagram classes include

- *CapsuleView* on page 477
  - Public Attributes

    ShowAllPorts : Boolean on page 477

    SuppressPorts : Boolean on page 477
- *ClassDiagram* on page 477
  - Public Attributes

    ParentLogicalPackage : LogicalPackage on page 478
  - Public Operations

    AddAssociation (theAssociation : Association) : Boolean on page 478

    AddCapsule (theCapsule : Capsule) : Boolean on page 478

    AddClass (theClass : Class) : Boolean on page 479

    AddLogicalPackage (theCat : LogicalPackage) : Boolean on page 479

    AddProtocol (theProtocol : Protocol) : Boolean on page 480

    AddUseCase (theUseCase : UseCase) : Boolean on page 480

    GetAssociations () : AssociationCollection on page 481

    GetCapsuleView (theCapsule : Capsule) : CapsuleView on page 481

    GetCapsules () : CapsuleCollection on page 482

    GetClassView (theClass : Class) : ClassView on page 482

    GetClasses () : ClassCollection on page 483

    GetLogicalPackages () : LogicalPackageCollection on page 483

    GetProtocolView (theProtocol : Protocol) : ProtocolView on page 484

    GetProtocols () : ProtocolCollection on page 484

    GetSelectedCapsules () : CapsuleCollection on page 485

    GetSelectedClasses () : ClassCollection on page 485

    GetSelectedLogicalPackages () : LogicalPackageCollection on page 485

    GetSelectedProtocols () : ProtocolCollection on page 486

GetUseCases () : UseCaseCollection on page 486

IsUseCaseDiagram () : Boolean on page 487

RemoveAssociation (theAssociation : Association) : Boolean on page 487

RemoveCapsule (theCapsule : Capsule) : Boolean on page 488

RemoveClass (theClass : Class) : Boolean on page 488

RemoveLogicalPackage (theLogicalPackage : LogicalPackage) : Boolean on page 489

RemoveProtocol (theProtocol : Protocol) : Boolean on page 489

RemoveUseCase (theUseCase : UseCase) : Boolean on page 490

- *ClassView* on page 490
- *ClassifierView* on page 490
  - Public Attributes

    AutomaticResize : Boolean on page 491

    ShowAllAttributes : Boolean on page 491

    ShowAllOperations : Boolean on page 491

    ShowCompartmentStereotypes : Boolean on page 491

    ShowOperationSignature : Boolean on page 491

    ShowVisibility : Boolean on page 492

    SuppressAttributes : Boolean on page 492

    SuppressOperations : Boolean on page 492
- *ProtocolView* on page 492
  - Public Attributes

    ShowAllInSignals : Boolean on page 492

    ShowAllOutSignals : Boolean on page 493

    SuppressInSignals : Boolean on page 493

    SuppressOutSignals : Boolean on page 493

CapsuleView

Description

The CapsuleView is the view elements representing capsules. CapsuleView allows changing the visibility of ports.

Derived from ClassifierView

Public Attributes

ShowAllPorts : Boolean

Description

Indicates whether the capsule's ports will be visible when the capsule view is displayed.

SuppressPorts : Boolean

Description

Indicates whether to suppress the capsule's ports compartment when the capsule view is displayed.

ClassDiagram

Description

The class diagram class allows you to add, retrieve and delete classes and categories to and from a class diagram. The class diagram class has a set of attributes and operations that apply specifically to class diagrams. In addition, it inherits all diagram class attributes and operations.

Check the lists of attributes and operations for complete information.

Derived from Diagram

Public Attributes

ParentLogicalPackage : LogicalPackage

Description

Specifies the LogicalPackage that contains the class diagram.

Public Operations

AddAssociation (theAssociation : Association) : Boolean

Description

Adds an association icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddAssociation (theAssociation)
```

```
Added As Boolean
```

Returns a value of True when the association icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the association icon is being added.

```
theAssociation As RoseRT.Association
```

Association whose icon is being added to this class diagram.

AddCapsule (theCapsule : Capsule) : Boolean

Description

Adds a capsule icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddCapsule(theCapsule)
```

```
Added As Boolean
```

Returns a value of True when the capsule icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the capsule icon is being added.

```
theCapsule As RoseRT.Capsule
```

Capsule whose icon is being added to this class diagram.

AddClass (theClass : Class) : Boolean

Description

Adds a class icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddClass (theClass)
```

```
Added As Boolean
```

Returns a value of True when the class icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the class icon is being added.

```
theClass As RoseRT.Class
```

Class whose icon is being added to this class diagram.

AddLogicalPackage (theCat : LogicalPackage) : Boolean

Description

adds a LogicalPackage icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddLogicalPackage (theLogicalPackage)
```

```
Added As Boolean
```

Returns a value of True when the LogicalPackage icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the LogicalPackage icon is being added.

```
theLogicalPackage As RoseRT.LogicalPackage
```

LogicalPackage whose icon is being added to the diagram.

AddProtocol (theProtocol : Protocol) : Boolean

Description

Adds a protocol icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddProtocol(theProtocol)
```

```
Added As Boolean
```

Returns a value of True when the protocol icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the protocol icon is being added.

```
theProtocol As RoseRT.Protocol
```

Protocol whose icon is being added to this class diagram.

AddUseCase (theUseCase : UseCase) : Boolean

Description

Adds a use case icon to a class diagram.

Syntax

```
Added = theClassDiagram.AddUseCase (theUseCase)
```

```
Added As Boolean
```

Returns a value of True when the use case icon is added to the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram to which the use case icon is being added.

```
theUseCase As RoseRT.UseCase
```

Use case whose icon is being added to the diagram.

GetAssociations () : AssociationCollection

Description

Retrieves a collection of associations from a class diagram.

Syntax

```
Set theAssociations = theClassDiagram.GetAssociations ( )
```

```
theAssociations As RoseRT.AssociationCollection
```

Returns the collection of associations from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the associations.

GetCapsuleView (theCapsule : Capsule) : CapsuleView

Description

Retrieves a capsule view from a class diagram. If the view does not yet exist, the method creates the view.

Syntax

```
Set theCapsuleView = theClassDiagram.GetCapsuleView (theCapsule)
```

```
theCapsuleView As RoseRT.CapsuleView
```

Returns a capsule view from a class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the capsule view.

```
theCapsule As RoseRT.Capsule
```

Capsule whose view is being retrieved.

GetCapsules () : CapsuleCollection

Description

Retrieves a collection of capsules from a class diagram.

Syntax

```
Set theCapsules = theClassDiagram.GetCapsules ( )
```

```
theCapsules As RoseRT.CapsuleCollection
```

Returns the collection of capsules from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the capsules.

GetClassView (theClass : Class) : ClassView

Description

Retrieves a class view from a class diagram. If the view does not yet exist, the method creates the view.

Syntax

```
Set theClassView = theClassDiagram.GetClassView (theClass)
```

```
theClassView As RoseRT.ClassView
```

Returns a class view from a class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the class view.

```
theClass As RoseRT.Class
```

Class whose view is being retrieved.

GetClasses () : ClassCollection

Description

Retrieves a collection of classes from a class diagram.

Syntax

```
Set theClasses = theClassDiagram.GetClasses ( )
```

```
theClasses As RoseRT.ClassCollection
```

Returns the collection of classes from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the classes.

GetLogicalPackages () : LogicalPackageCollection

Description

Retrieves a collection of LogicalPackages from a class diagram.

Syntax

```
Set theLogicalPackages = theClassDiagram.GetLogicalPackages ( )
```

```
theLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns the collection of LogicalPackages from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the LogicalPackages.

GetProtocolView (theProtocol : Protocol) : ProtocolView

Description

Retrieves a protocol view from a class diagram. If the view does not yet exist, the method creates the view.

Syntax

```
Set theProtocolView = theClassDiagram.GetProtocolView (theProtocol)
```

```
theProtocolView As RoseRT.ProtocolView
```

Returns a protocol view from a class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the protocol view.

```
theProtocol As RoseRT.Protocol
```

Protocol whose view is being retrieved.

GetProtocols () : ProtocolCollection

Description

Retrieves a collection of protocols from a class diagram.

Syntax

```
Set theProtocols = theClassDiagram.GetProtocols ( )
```

```
theProtocols As RoseRT.ProtocolCollection
```

Returns the collection of protocols from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the protocols.

GetSelectedCapsules () : CapsuleCollection

Description

Retrieves the collection of currently selected capsules from a class diagram.

Syntax

```
Set theCapsules = theClassDiagram.GetSelectedCapsules ( )
```

`theCapsules As RoseRT.CapsuleCollection`

Returns the collection of currently selected capsules from the classes diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Class diagram from which to retrieve the capsules.

GetSelectedClasses () : ClassCollection

Description

Retrieves the collection of currently selected classes from a class diagram.

Syntax

```
Set theClasses = theClassDiagram.GetSelectedClasses ( )
```

`theClasses As RoseRT.ClassCollection`

Returns the collection of currently selected classes from the classes diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Class diagram from which to retrieve the classes.

GetSelectedLogicalPackages () : LogicalPackageCollection

Description

Retrieves the collection of currently selected LogicalPackages from a class diagram.

Syntax

```
Set theLogicalPackages = theClassDiagram.GetSelectedLogicalPackages (
)
```

```
theLogicalPackages As RoseRT.LogicalPackageCollection
```

Returns the collection of currently selected LogicalPackages from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the LogicalPackages.

GetSelectedProtocols () : ProtocolCollection

Description

Retrieves the collection of currently selected protocols from a class diagram.

Syntax

```
Set theProtocols = theClassDiagram.GetSelectedProtocols ( )
```

```
theProtocols As RoseRT.ProtocolCollection
```

Returns the collection of currently selected protocols from the classes diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the protocols.

GetUseCases () : UseCaseCollection

Description

Retrieves a collection of use cases from a class diagram.

Syntax

```
Set theUseCases = theClassDiagram.GetUseCases ( )
```

```
theUseCases As RoseRT.UseCaseCollection
```

Returns the collection of use cases from the class diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Class diagram from which to retrieve the use cases.

IsUseCaseDiagram () : Boolean

Description

Determines whether a class diagram is a use case diagram.

Syntax

```
IsUseCase = theClassDiagram.IsUseCaseDiagram ( )
```

```
IsUseCase As Boolean
```

Returns a value of True if the specified class diagram is a use case diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

The instance of the class diagram being tested as a use case diagram.

RemoveAssociation (theAssociation : Association) : Boolean

Description

Removes an association icon from a class diagram.

Syntax

```
Removed = theClassDiagram.RemoveAssociation (theAssociation)
```

```
Removed As Boolean
```

Returns a value of True when the association icon is removed from the diagram.

```
theClassDiagram As RoseRT.ClassDiagram
```

Diagram from which the association icon is being removed.

```
theAssociation As RoseRT.Association
```

Association whose icon is being removed from the diagram.

RemoveCapsule (theCapsule : Capsule) : Boolean

Description

Removes a capsule icon from a class diagram.

Syntax

```
Removed = theClassDiagram.RemoveCapsule (theCapsule)
```

`Removed As Boolean`

Returns a value of True when the capsule icon is removed from the diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Diagram from which the capsule icon is being removed.

`theCapsule As RoseRT.Capsule`

Capsule whose icon is being removed from the diagram.

RemoveClass (theClass : Class) : Boolean

Description

Removes a class icon from a class diagram.

Syntax

Removed = theClassDiagram.RemoveClass (theClass)

`Removed As Boolean`

Returns a value of True when the class icon is removed from the diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Diagram from which the class icon is being removed.

`theClass As RoseRT.Class`

Class whose icon is being removed from the diagram.

RemoveLogicalPackage (theLogicalPackage : LogicalPackage) : Boolean

Description

Removes a LogicalPackage icon from a class diagram.

Syntax

```
Removed = theClassDiagram.RemoveLogicalPackage (theLogicalPackage)
```

`Removed As Boolean`

Returns a value of True when the LogicalPackage icon is removed from the diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Diagram from which the LogicalPackage icon is being removed.

`theLogicalPackage As RoseRT.LogicalPackage`

LogicalPackage whose icon is being removed from the diagram.

RemoveProtocol (theProtocol : Protocol) : Boolean

Description

Removes a protocol icon from a class diagram.

Syntax

```
Removed = theClassDiagram.RemoveProtocol (theProtocol)
```

`Removed As Boolean`

Returns a value of True when the protocol icon is removed from the diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Diagram from which the protocol icon is being removed.

`theProtocol As RoseRT.Protocol`

Protocol whose icon is being removed from the diagram.

RemoveUseCase (theUseCase : UseCase) : Boolean

Description

Removes a use case icon from a class diagram.

Syntax

```
Removed = theClassDiagram.RemoveUseCase (theUseCase)
```

`Removed As Boolean`

Returns a value of True when the use case icon is removed from the diagram.

`theClassDiagram As RoseRT.ClassDiagram`

Diagram from which the use case icon is being removed.

`theUseCase As RoseRT.UseCase`

Use case whose icon is being removed from the diagram.

ClassView

Description

The ClassView is the view elements representing classes. ClassView allows changing the visibility of attributes and operations.

Derived from ClassifierView

ClassifierView

Description

The ClassifierView is the base class of the view elements representing classifiers. ClassifierView allows changing the visibility of different common classifier features such as attributes and operations.

Derived from ViewElement

Public Attributes

AutomaticResize : Boolean

Description

Indicates whether the class view will be automatically resized when displayed in the view port. Corresponds to the Automatic Resize option in Rose context menus.

ShowAllAttributes : Boolean

Description

Indicates whether the class's attributes will be visible when the class view is displayed in the view port

ShowAllOperations : Boolean

Description

Indicates whether the class's operations will be visible when the class view is displayed in the view port. Corresponds to the Show All Operations option in Rose context menus.

ShowCompartmentStereotypes : Boolean

Description

Indicates whether to show stereotypes of features in compartments when the classifier view is displayed.

ShowOperationSignature : Boolean

Description

Indicates whether the class's operations signature will be shown when the class view is displayed in the view port. Corresponds to the Show Operations Signature option in Rose context menus.

ShowVisibility : Boolean

Description

Indicates whether to show the classifier visibility when the classifier view is displayed.

SuppressAttributes : Boolean

Description

Indicates whether to suppress the class's attributes compartment when the class view is displayed in the view port.

SuppressOperations : Boolean

Description

Indicates whether to suppress the class's operations compartment when the class view is displayed in the view port.

ProtocolView

Description

The ProtocolView is the view elements representing protocols. ProtocolView allows changing the visibility of signals.

Derived from ClassifierView

Public Attributes

ShowAllInSignals : Boolean

Description

Indicates whether protocol's in signals will be visible when the protocol view is displayed.

ShowAllOutSignals : Boolean

Description

Indicates whether protocol's out signals will be visible when the protocol view is displayed.

SuppressInSignals : Boolean

Description

Indicates whether to suppress the protocol's in signals compartment when the protocol view is displayed.

SuppressOutSignals : Boolean

Description

Indicates whether to suppress the protocol's out signals compartment when the protocol view is displayed.

Collaboration Diagram Classes

Collaboration Diagram classes include

- *CapsuleRoleView* on page 494
  - Public Attributes

    EditingInside : Boolean on page 495

    PositionBySuperClass : Boolean on page 495
  - Public Operations

    AutoAdjustConnectors () : on page 495

    GoInside () : on page 495
- *CollaborationDiagram* on page 496
  - Public Operations

    AddAssociationRoleView (pAssocRole : AssociationRole) : ViewElement on page 496

    AddCapsuleRoleView (pCapsulerRole : CapsuleRole) : CapsuleRoleView on page 496

AddClassifierRoleView (pClassifierRole : ClassifierRole) : ClassifierRoleView on page 497

AddConnectorView (pConnector : Connector) : ViewElement on page 498

AddPortView (pPort : Port) : PortView on page 498

- *PortRoleView* on page 499
  - Public Attributes

    AutoAdjustOn : Boolean on page 499

    CapsuleRoleView : ViewElement on page 499

    PositionBySuperClass : Boolean on page 499
  - Public Operations

    AutoAdjust () : on page 499
- *PortView* on page 500
  - Public Attributes

    PositionBySuperClass : Boolean on page 500

    StructurePerimeterView : ViewElement on page 500
- *StructurePerimeterView* on page 500
  - Public Attributes

    PositionBySuperClass : Boolean on page 501

CapsuleRoleView

Description

CapsuleRoleView contains properties and methods that define the appearance of a Capsule Role within a structure (collaboration) diagram.

Derived from ViewElement

Public Attributes

EditingInside : Boolean

Description

Whether a user is allowed to directly edit the inside of a capsule role that appears on a structure diagram.

PositionBySuperClass : Boolean

Description

Whether the CapsuleRoleView inherits its position information from that of its superclass.

Public Operations

AutoAdjustConnectors () :

Description

Allows connectors to auto adjust themsleves to the shortest path between an originating and a destination capsule role.

Syntax

```
theCapsuleRoleView.AutoAdjustConnectors()


theCapsuleRoleView As RoseRT.CapsuleRoleView
```

The capsule role view to adjust connectors to.

GoInside () :

Description

Open the structure diagram that represents the inside of the CapsuleRole.

Syntax

```
theCapsuleRoleView.GoInside()
```

```
theCapsuleRoleView As RoseRT.CapsuleRoleView
```

The capsule role view to go inside.

CollaborationDiagram

Description

CollaborationDiagram graphically shows the capsule roles, and ports contained within a Collaboration (Structure) Diagram.

Derived from Diagram

Public Operations

AddAssociationRoleView (pAssocRole : AssociationRole) : ViewElement

Description

Add a ViewElement that represents an association role to the inside of a Collaboration Diagram.

Syntax

```
Set theViewElement =
theCollaborationDiagram.AddAssociationRoleView(pAssocRole)
```

```
theViewElement As RoseRT.ViewElement
```

Returns the view object being added to the diagram.

```
theCollaborationDiagram As RoseRT.CollaborationDiagram
```

CollaborationDiagram to which the object is being added.

```
pAssocRole As RoseRT.AssociationRole
```

The Association Role for which a view object is being added.

AddCapsuleRoleView (pCapsulerRole : CapsuleRole) : CapsuleRoleView

Description

Add a CapsuleRoleView to the inside of a Collaboration Diagram.

Syntax

```
Set theCapRoleView =
theCollaborationDiagram.AddCapsuleRoleView(pCapRole)
```

```
theCapRoleView As RoseRT.CapsuleRoleView
```

Returns the view object being added to the diagram.

```
theCollaborationDiagram As RoseRT.CollaborationDiagram
```

CollaborationDiagram to which the object is being added.

```
pCapRole As RoseRT.CapsuleRole
```

The CapsuleRole for which a view object is being added.

AddClassifierRoleView (pClassifierRole : ClassifierRole) : ClassifierRoleView

Description

Add a ClassifierRoleView to the inside of a Collaboration Diagram.

Syntax

```
Set theClassRoleView = theCollaborationDiagram.AddClassifierRoleView
(pClassRole)
```

```
theClassRoleView As RoseRT.ClassifierRoleView
```

Returns the view object being added to the diagram.

```
theCollaborationDiagram As RoseRT.CollaborationDiagram
```

CollaborationDiagram to which the object is being added.

```
pClassRole As RoseRT.ClassifierRole
```

The ClassifierRole for which a view object is being added.

AddConnectorView (pConnector : Connector) : ViewElement

Description

Add a ViewElement that represents an connector to the inside of a Collaboration Diagram.

Syntax

```
Set theViewElement =
theCollaborationDiagram.AddConnectorView(pConnector)
```

`theViewElement As RoseRT.ViewElement`

Returns the view object being added to the diagram.

`theCollaborationDiagram As RoseRT.CollaborationDiagram`

CollaborationDiagram to which the object is being added.

`pConnector As RoseRT.Connector`

The Connector for which a view object is being added.

AddPortView (pPort : Port) : PortView

Description

Add a PortView to the inside of a Collaboration Diagram.

Syntax

```
Set thePortView = theCollaborationDiagram.AddPortView(pPort)
```

`thePortView As RoseRT.PortView`

Returns the view object being added to the diagram.

`theCollaborationDiagram As RoseRT.CollaborationDiagram`

CollaborationDiagram to which the object is being added.

`pPort As RoseRT.Port`

The Port for which a view object is being added.

PortRoleView

Description

PortRoleView contains properties and methods that define the appearance of a Port Role within a structure (collaboration) diagram. A Port Role is a port bound to a capsule role.

Derived from ViewElement

Public Attributes

AutoAdjustOn : Boolean

Description

Whether AutoAdjust has been selected.

CapsuleRoleView : ViewElement

Description

The CapsuleRoleView to which the PortRoleView is bound.

PositionBySuperClass : Boolean

Description

Whether the PortRoleView inherits its position information from that of its superclass.

Public Operations

AutoAdjust () :

Description

Allows connectors to auto adjust themsleves to the shortest path between port roles.

Syntax

```
thePortRoleView.AutoAdjust()
```

```
thePortRoleView As RoseRT.PortRoleView
```

The port role view to auto adjust.

PortView

Description

PortView contains properties and methods that define the appearance of a Port within a structure (collaboration) diagram.

Derived from ViewElement

Public Attributes

PositionBySuperClass : Boolean

Description

Whether the PortView inherits its position information from that of its superclass.

StructurePerimeterView : ViewElement

Description

If it's a public port, this is the StructurePerimeterView that this PortView is bound to. If it's a protected port, the PortView is not bound to a StructurePerimeterView.

StructurePerimeterView

Description

StructurePerimeterView contains properties and methods that define the appearance of the outer boundary shown in a collaboration (structure) diagram.

Derived from ViewElement

Public Attributes

PositionBySuperClass : Boolean

Description

Whether the StructurePerimeterView inherits its position information from that of its superclass.

Component Diagram Classes

Component Diagram classes include

- *ComponentDiagram* on page 502
  - Public Attributes

    ComponentPackageViews : ComponentPackageViewCollection on page 502

    ComponentViews : ComponentViewCollection on page 502

    ParentComponentPackage : ComponentPackage on page 502
  - Public Operations

    AddComponent (theMod : Component) : Boolean on page 503

    AddComponentPackage (theComponentPackage : ComponentPackage) : Boolean on page 503

    AddComponentPackageView (aComponentPackage : ComponentPackage) : ComponentPackageView on page 504

    AddComponentView (aComponent : Component) : ComponentView on page 504

    GetComponentPackages () : ComponentPackageCollection on page 505

    GetComponents () : ComponentCollection on page 505

    GetSelectedComponentPackages () : ComponentPackageCollection on page 506

    GetSelectedComponents () : ComponentCollection on page 506

    RemoveComponentPackageView (aComponentPackageView : ComponentPackageView) : Boolean on page 506

    RemoveComponentView (aComponentView : ComponentView) : Boolean on page 507

- *ComponentPackageView* on page 508
  - Public Operations

    GetComponentPackage () : ComponentPackage on page 508
- *ComponentView* on page 509
  - Public Operations

    GetComponent () : Component on page 509

ComponentDiagram

Description

A component diagram maps the allocation classes and objects to components. The component diagram class exposes attributes and operations that allow you to add, retrieve, and delete classes and objects in a component diagram.

Check the lists of attributes and operations for complete information.

Derived from Diagram

Public Attributes

ComponentPackageViews : ComponentPackageViewCollection

Description

The collection of ComponentPackageView shown in a component diagram.

ComponentViews : ComponentViewCollection

Description

The collection of ComponentView shown in a component diagram.

ParentComponentPackage : ComponentPackage

Description

Identifies the ComponentPackage object that contains the component and is always set to a valid object (is never set to Nothing)

Public Operations

AddComponent (theMod : Component) : Boolean

Description

Adds a component icon to a component diagram.

Syntax

```
Added = theDiagram.AddComponent (theComponent)
```

`Added As Boolean`

Returns a value of True when the component is added.

`theDiagram As RoseRT.ComponentDiagram`

Component diagram to which the component is being added.

`theComponent As RoseRT.Component`

Component being added to the diagram.

AddComponentPackage (theComponentPackage : ComponentPackage) : Boolean

Description

Adds the view associated with a ComponentPackage to a component diagram.

Syntax

```
Added = theDiagram.AddComponentPackage (theComponentPackage)
```

`Added As Boolean`

Returns a value of True when the view associated with a ComponentPackage is added.

`theDiagram As RoseRT.ComponentDiagram`

Component diagram to which the ComponentPackageView is being added.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage whose associated view is being added to the diagram.

AddComponentPackageView (aComponentPackage : ComponentPackage) : ComponentPackageView

Description

Adds the view associated with a ComponentPackage to a component diagram.

Syntax

```
Added = theDiagram.AddComponentPackageView (theComponentPackage)
```

```
Added As Boolean
```

Returns a value of True when the view associated with a ComponentPackage is added.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram to which the ComponentPackageView is being added.

```
theComponentPackage As RoseRT.ComponentPackage
```

ComponentPackage whose associated view is being added to the diagram.

AddComponentView (aComponent : Component) : ComponentView

Description

Adds a ComponentView to a component diagram.

Syntax

```
Added = theDiagram.AddComponentView ( aComponent )
```

```
Added As Boolean
```

Returns a value of True when the ComponentView is added.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram to which the ComponentPackage is being added.

```
aComponent As RoseRT.Component
```

Component whose view is being added to the diagram.

GetComponentPackages () : ComponentPackageCollection

Description

Retrieves the collection of the ComponentPackages associated with each of the ComponentPackageViews shown in a component diagram.

Syntax

```
Set theComponentPackages = theDiagram.GetComponentPackages()
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns the collection of the ComponentPackages.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram whose ComponentPackages are being retrieved.

GetComponents () : ComponentCollection

Description

Retrieves the collection of the Components associated with each of the ComponentViews shown in a component diagram.

Syntax

```
Set theComponents = theDiagram.GetComponents()
```

```
theComponents As RoseRT.ComponentCollection
```

Returns the collection of the Components.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram whose Components are being retrieved.

GetSelectedComponentPackages () : ComponentPackageCollection

Description

Retrieves the collection of currently selected component packages from a component diagram.

Syntax

```
Set theComponentPackagess = theDiagram.GetSelectedComponentPackages (
)
```

```
theComponentPackages As RoseRT.ComponentPackageCollection
```

Returns the collection of currently selected component packages from the component diagram.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram from which to retrieve the component packages.

GetSelectedComponents () : ComponentCollection

Description

Retrieves the collection of currently selected components from a component diagram.

Syntax

```
Set theComponents = theDiagram.GetSelectedComponents ( )
```

```
theComponents As RoseRT.ComponentCollection
```

Returns the collection of currently selected components from the component diagram.

```
theDiagram As RoseRT.ComponentDiagram
```

Component diagram from which to retrieve the components.

RemoveComponentPackageView (aComponentPackageView : ComponentPackageView) : Boolean

Description

Removes a ComponentPackageView from a component diagram.

Syntax

```
Removed = theDiagram.RemoveComponentPackageView
(aComponentPackageView)
```

`Removed As Boolean`

Returns a value of True when the ComponentPackageView is successfully removed from the diagram.

`theDiagram As RoseRT.ComponentDiagram`

Component diagram from which the ComponentPackageView is being removed.

`aComponentPackageView As RoseRT.ComponentPackageView`

ComponentPackageView being removed from the diagram.

RemoveComponentView (aComponentView : ComponentView) : Boolean

Description

Removes a ComponentView from a component diagram.

Syntax

```
Removed = theDiagram.RemoveComponentView (aComponentView)
```

`Removed As Boolean`

Returns a value of True when the ComponentView is successfully removed from the diagram.

`theDiagram As RoseRT.ComponentDiagram`

Component diagram from which the ComponentView is being removed.

`aComponentView As RoseRT.ComponentView`

ComponentView being removed from the diagram.

ComponentPackageView

Description

ComponentPackages contain components, as well as other ComponentPackages. The ComponentPackage view is the visual representation of a ComponentPackage, and is what appears on a diagram in the model. The ComponentPackage view class inherits the ViewElement attributes and operations that determine the size and placement of the ComponentPackage view. It also allows you to retrieve the ComponentPackage object itself from the ComponentPackage view.

Check the lists of attributes and operations for complete information.

Derived from ViewElement

Public Operations

GetComponentPackage () : ComponentPackage

Description

Retrieves the ComponentPackage represented by the ComponentPackage view.

Syntax

```
Set theComponentPackage = theComponentPackageView.GetObject ( )
```

```
theComponentPackage As RoseRT.ComponentPackage
```

Returns the ComponentPackage represented by the ComponentPackageview. Note that the REI return class is currently called component, not ComponentPackage.

```
theComponentPackageView As RoseRT.ComponentPackageView
```

Instance of the ComponentPackage view whose corresponding ComponentPackage (component) is being retrieved.

ComponentView

Description

ComponentView contains properties and methods that define the appearance of a Component within a component diagram.

Derived from ViewElement

Public Operations

GetComponent () : Component

Description

Gets the Component associated with this ComponentView.

Syntax

```
Set theComponent = theComponentView.GetComponent()
```

```
theComponent As RoseRT.Component
```

Returns the component.

```
theComponentView As RoseRT.ComponentView
```

ComponentView from which to get the component.

Deployment Diagram Classes

Deployment Diagram classes include

- *DeploymentDiagram* on page 510
    - Public Operations

      AddDevice (theDevice : Device, x : Integer, y : Integer) : ViewElement on page 510

      AddProcessor (theProcessor : Processor, x : Integer, y : Integer) : ViewElement on page 511

      GetDevices () : DeviceCollection on page 512

      GetProcessors () : ProcessorCollection on page 512

RemoveDevice (theDevice : Device) : Boolean on page 512

RemoveProcessor (theProcessor : Processor) : Boolean on page 513

DeploymentDiagram

Description

A deployment diagram is a visual representation of devices and processors. The deployment diagram class exposes properties and methods that allow you to add, retrieve and delete devices and processors in a deployment diagram.

Check the lists of attributes and operations for complete information.

Derived from Diagram

Public Operations

AddDevice (theDevice : Device, x : Integer, y : Integer) : ViewElement

Description

Adds a device icon to a deployment diagram.

Syntax

```
Set theView = theDeploymentDiagram.AddDevice (theDevice, XPosition,
YPosition)
```

`theView As RoseRT.ModelElementView`

Returns the device icon being added to the diagram.

`theDeploymentDiagram As RoseRT.DeploymentDiagram`

Diagram to which the icon is being added.

`theDevice As RoseRT.Device`

Device whose icon is being added to the diagram.

```
Xposition As Integer
```

X axis coordinate of the icon in the diagram.

```
YPosition As Integer
```

Y axis coordinate of the icon in the diagram.

AddProcessor (theProcessor : Processor, x : Integer, y : Integer) : ViewElement

Description

Adds a processor icon to a deployment diagram.

Syntax

```
Set theView = theDeploymentDiagram.AddProcessor (theProcessor,
XPosition, YPosition)
```

```
theView As RoseRT.ModelElementView
```

Returns the processor icon being added to the diagram.

```
theDeploymentDiagram As RoseRT.DeploymentDiagram
```

Diagram to which the icon is being added.

```
theProcessor As RoseRT.Processor
```

Processor whose icon is being added to the diagram.

```
XPosition As Integer
```

X axis coordinate of the icon in the diagram.

```
YPosition As Integer
```

Y axis coordinate of the icon in the diagram.

GetDevices () : DeviceCollection

Description

Retrieves the collection of devices belonging to the deployment diagram.

Syntax

```
Set theDevices = theDeploymentDiagram.GetDevices ( )
```

`theDevices As RoseRT.DeviceCollection`

Returns the collection of devices belonging to the deployment diagram.

`theDeploymentDiagram As RoseRT.DeploymentDiagram`

Deployment diagram from which to retrieve the devices.

GetProcessors () : ProcessorCollection

Description

Retrieves the collection of processors belonging to the deployment diagram.

Syntax

```
Set theProcessors = theDeploymentDiagram.GetProcessors ( )
```

`theProcessors As RoseRT.ProcessorCollection`

Returns the collection of processors belonging to the deployment diagram.

`theDeploymentDiagram As RoseRT.DeploymentDiagram`

Deployment diagram from which to retrieve the processors.

RemoveDevice (theDevice : Device) : Boolean

Description

Removes a device icon from a deployment diagram.

Syntax

```
Removed = theDeploymentDiagram.RemoveDevice (theDevice)
```

```
Removed As Boolean
```
Returns a value of True when the device icon is removed.

```
theDeploymentDiagram As RoseRT.DeploymentDiagram
```
Diagram from which the icon is being removed.

```
theDevice As RoseRT.Device
```
Device whose icon is being removed from the diagram.

RemoveProcessor (theProcessor : Processor) : Boolean

Description

Removes a processor icon from a deployment diagram.

Syntax

```
Removed = theDeploymentDiagram.RemoveProcessor (theProcessor)
```

```
Removed As Boolean
```
Returns a value of True when the processor icon is removed.

```
theDeploymentDiagram As RoseRT.DeploymentDiagram
```
Diagram from which the icon is being removed.

```
theProcessor As RoseRT.Processor
```
Processor whose icon is being removed from the diagram.

Sequence Diagram Classes

Sequence Diagram classes

- *ClassifierRoleView* on page 514
- *CreateMessageView* on page 514

- *InteractionInstanceView* on page 515
    - Public Attributes

      CreateMessageView : MessageView on page 515

      DestroyMessageView : MessageView on page 515
- *LifeLineView* on page 515
    - Public Attributes

      InteractionInstanceView : InteractionInstanceView on page 516
- *MessageView* on page 516
    - Public Attributes

      FromInstanceView : InteractionInstanceView on page 516

      ToInstanceView : InteractionInstanceView on page 516
- *SequenceDiagram* on page 516

ClassifierRoleView

Description

ClassifierRoleView contains properties and methods that define the appearance of a Classifier Role on a collaboration diagram.

Derived from ViewElement

CreateMessageView

Description

CreateMessageView contains properties and methods that define the appearance of a Create Message within a sequence diagram.

Derived from ViewElement

InteractionInstanceView

Description

InteractionInstanceView contains properties and methods that define the appearance of an interaction instance within a sequence diagram.

Derived from ViewElement

Public Attributes

CreateMessageView : MessageView

Description

The MessageView representing the optional Create Message for this interaction instance.

DestroyMessageView : MessageView

Description

The MessageView representing the optional Destroy Message for this interaction instance.

LifeLineView

Description

LifeLineView contains properties and methods that define the appearance of the Life Line of an interaction instance within a sequence diagram. The life line is the line that descends from the interaction instance rectangle.

Derived from ViewElement

Public Attributes

InteractionInstanceView : InteractionInstanceView

Description

The InteractionInstanceView associated with this LifeLineView.

MessageView

Description

MessageView contains properties and methods that define the appearance of a Message within a sequence diagram.

Derived from ViewElement

Public Attributes

FromInstanceView : InteractionInstanceView

Description

The InteractionInstanceView that represents the originator of the message.

ToInstanceView : InteractionInstanceView

Description

The InteractionInstance that represents the destination of the message.

SequenceDiagram

Description

SequenceDiagram graphically shows the interaction instances, messages, local states, local actions and coregions contained within a Sequence Diagram.

Derived from Diagram

State Diagram Classes

State Diagram classes include

- *BranchPointView* on page 519
    - Public Attributes

      BranchView : ChoicePointView on page 519
- *ChoicePointView* on page 519
    - Public Attributes

      Angle : Double on page 519

      BranchPointViewFalse : BranchPointView on page 519

      BranchPointViewIn : BranchPointView on page 520

      BranchPointViewTrue : BranchPointView on page 520

      Flipped : Boolean on page 520

      PositionBySuperClass : Boolean on page 520

      AutoAdjustTransitions () : on page 520
- *CompositeStateView* on page 521
    - Public Attributes

      EditingInside : Boolean on page 521

      PositionBySuperClass : Boolean on page 521

      SubDiagram : StateDiagram on page 521
    - Public Operations

      AutoAdjustTransitions () : on page 521

      GoInside () : on page 522
- *CoregionView* on page 522
- *FinalStateView* on page 522
    - Public Attributes

      PositionBySuperClass : Boolean on page 523

- *InitialPointView* on page 523
    - Public Attributes

      PositionBySuperClass : Boolean on page 523
- *JunctionAdornmentView* on page 523
    - Public Attributes

      JunctionView : JunctionPointView on page 524
- *JunctionPointView* on page 524
    - Public Attributes

      AutoAdjustOn : Boolean on page 524

      CompositeStateView : CompositeStateView on page 524

      JunctionAdornmentView : JunctionAdornmentView on page 524

      PositionBySuperClass : Boolean on page 525
    - Public Operations

      AutoAdjust () : on page 525
- *LocalStateOrActionView* on page 525
- *StateDiagram* on page 525
    - Public Operations

      AddChoicePointView (pChoicePoint : ChoicePoint) : ChoicePointView on page 526

      AddFinalStateView (pFinal : FinalState) : FinalStateView on page 526

      AddStateView (pState : CompositeState) : CompositeStateView on page 527
- *StatePerimeterView* on page 527
    - Public Attributes

      PositionBySuperClass : Boolean on page 528

BranchPointView

Description

Each ChoicePointView contains three BranchPointView elements representing the incoming state transition, and the outgoing true and false state transitions.

Derived from ViewElement

Public Attributes

BranchView : ChoicePointView

Description

The ChoicePointView to which this BranchPointView belongs.

ChoicePointView

Description

A Choice Point encapsulates action code that returns a conditional value of True or False. ChoicePointView contains properties and methods that define the appearance of a Choice Point within a state diagram.

Derived from ViewElement

Public Attributes

Angle : Double

Description

Rotation angle of the ChoicePointView, expressed in radians.

BranchPointViewFalse : BranchPointView

Description

Identifies the BranchPointView located at the outgoing False state transition.

BranchPointViewIn : BranchPointView

Description

Identifies the BranchPointView located at the incoming state transition.

BranchPointViewTrue : BranchPointView

Description

Identifies the BranchPointView located at the outgoing True state transition.

Flipped : Boolean

Description

Whether the ChoicePoint is shown flipped on the diagram.

PositionBySuperClass : Boolean

Description

Whether the ChoicePointView inherits its position information from the ChoicePointView in the state diagram of its superclass.

Public Operations

AutoAdjustTransitions () :

Description

Allows transitions to auto adjust themsleves to the shortest path between a choice point and an originating or destination state.

Syntax

```
theChoicePointView.AutoAdjustTransitions()
```

```
theChoicePointView As RoseRT.ChoicePointView
```

The choice point view to auto adjust.

CompositeStateView

Description

A CompositeState is a normal State as found on state diagrams. CompositeStateView contains properties and methods that define the appearance of a CompositeState within a state diagram.

Derived from ViewElement

Public Attributes

EditingInside : Boolean

Description

Whether a user is allowed to directly edit the inside of a state that appears on a state diagram.

PositionBySuperClass : Boolean

Description

Whether the CompositeStateView inherits its position information from that of its superclass.

SubDiagram : StateDiagram

Description

The state diagram that represents the inside of the CompositeState.

Public Operations

AutoAdjustTransitions () :

Description

Allows transitions to auto adjust themsleves to the shortest path between an originating and a destination state.

Syntax

```
theCompositeStateView.AutoAdjustTransitions()
```

```
theCompositeStateView As RoseRT.CompositeStateView
```

The composite state view to auto adjust.

GoInside () :

Description

Open the state diagram that represents the inside of the CompositeState.

Syntax

```
theCompositeStateView.GoInside()
```

```
theCompositeStateView As RoseRT.CompositeStateView
```

The composite state view to go inside.

CoregionView

Description

CoregionView contains properties and methods that define the appearance of a Coregion within a sequence diagram.

Derived from ViewElement

FinalStateView

Description

FinalStateView contains properties and methods that define the appearance of a FinalState within a state diagram.

Derived from ViewElement

Public Attributes

PositionBySuperClass : Boolean

Description

Whether the FinalStateView inherits its position information from that of its superclass.

InitialPointView

Description

InitialPointView contains properties and methods that define the appearance of a InitialPoint within a state diagram.

Derived from ViewElement

Public Attributes

PositionBySuperClass : Boolean

Description

Whether the InitialPointView inherits its position information from that of its superclass.

JunctionAdornmentView

Description

JunctionAdornmentView contains properties and methods that define the appearance of a Junction Adornment within a state diagram.

Derived from ViewElement

Public Attributes

JunctionView : JunctionPointView

Description

The JunctionView associated with this JunctionAdornmentView.

JunctionPointView

Description

Composite States contain Junction Points where they join with incoming and outgoing state transitions. JunctionPointView contains properties and methods that define the appearance of a JunctionPoint within a state diagram.

Derived from ViewElement

Public Attributes

AutoAdjustOn : Boolean

Description

Whether AutoAdjust has been selected.

CompositeStateView : CompositeStateView

Description

The CompositeStateView associated with this JunctionPointView.

JunctionAdornmentView : JunctionAdornmentView

Description

The JunctionAdornmentView associated with this JunctionView.

PositionBySuperClass : Boolean

Description

Whether the JunctionPointView inherits its position information from that of its superclass.

Public Operations

AutoAdjust () :

Description

Allows transitions to auto adjust themsleves to the shortest path between an originating and a destination state.

Syntax

```
theJunctionPointView.AutoAdjust()
```

```
theJunctionPointView As RoseRT.JunctionPointView
```

The junction point view to auto adjust.

LocalStateOrActionView

Description

LocalStateOrActionView contains properties and methods that define the appearance of Local States and Local Actions within sequence diagrams.

Derived from ViewElement

StateDiagram

Description

A State Diagram graphically shows the states and transitions within the behavior of a capsule, class or use case.

Derived from Diagram

Public Operations

AddChoicePointView (pChoicePoint : ChoicePoint) : ChoicePointView

Description

Add a ChoicePointView to the inside of a State Diagram.

Syntax

```
Set theChoicePointView =
theStateDiagram.AddChoicePointView(pChoicePoint)
```

`theChoicePointView As RoseRT.ChoicePointView`

Returns the view object being added to the diagram.

`theStateDiagram As RoseRT.StateDiagram`

StateDiagram to which the object is being added.

`pChoicePoin As RoseRT.ChoicePoint`

The ChoicePoint for which a view object is being added.

AddFinalStateView (pFinal : FinalState) : FinalStateView

Description

Add a FinalStateView to the inside of a State Diagram.

Syntax

```
Set theFinalStateView = theStateDiagram.AddFinalStateView(pFinal)
```

`theFinalStateView As RoseRT.FinalStateView`

Returns the view object being added to the diagram.

`theStateDiagram As RoseRT.StateDiagram`

StateDiagram to which the object is being added.

```
pFinal As RoseRT.FinalState
```

The FinalState for which a view object is being added.

AddStateView (pState : CompositeState) : CompositeStateView

Description

Add a CompositeStateView to the inside of a State Diagram.

Syntax

```
Set theCompositeStateView = theStateDiagram.AddStateView(pState)
```

```
theCompositeStateView As RoseRT.CompositeStateView
```

Returns the view object being added to the diagram.

```
theStateDiagram As RoseRT.StateDiagram
```

StateDiagram to which the object is being added.

```
pState As RoseRT.CompositeState
```

The CompositeState for which a view object is being added.

StatePerimeterView

Description

StatePerimeterView contains properties and methods that define the appearance of the outer state boundary shown in a state diagram.

Derived from ViewElement

Public Attributes

PositionBySuperClass : Boolean

Description

Whether the StatePerimeterView inherits its position information from that of its superclass.

View Property Classes

View Property classes include

- *LineVertex* on page 529
    - Public Operations

      GetXPosition () : Integer on page 529

      GetYPosition () : Integer on page 530
- *View_FillColor* on page 530
    - Public Attributes

      Blue : Integer on page 530

      Green : Integer on page 530

      Red : Integer on page 531

      Transparent : Boolean on page 531
- *View_Font* on page 531
    - Public Attributes

      Blue : Integer on page 531

      Bold : Boolean on page 531

      FaceName : String on page 531

      Green : Integer on page 532

      Italic : Boolean on page 532

      Red : Integer on page 532

Size : Integer on page 532

StrikeThrough : Boolean on page 532

Underline : Boolean on page 532

- *View_LineColor* on page 532
  - Public Attributes

  Blue : Integer on page 533

  Green : Integer on page 533

  Red : Integer on page 533

LineVertex

Description

Represents a point of a line.

Derived from RRTEIObject

Public Operations

GetXPosition () : Integer

Description

Retrieves a vertex' X coordinate.

Syntax

```
X = theLineVertex.GetXPosition()
```

```
X As Integer
```

The X coordinate of the vertex.

```
theLineVertex As RoseRT.LineVertex
```

LineVertex from which to retrieve coordinate.

GetYPosition () : Integer

Description

Retrieves a vertex' Y coordinate.

Syntax

```
Y = theLineVertex.GetYPosition()
```

```
Y As Integer
```

The Y coordinate of the vertex.

```
theLineVertex As RoseRT.LineVertex
```

LineVertex from which to retrieve coordinate.

View_FillColor

Description

Specifies the amount of red, green, or blue to use in the fill color for the ModelElementView object, or whether it is transparent.

Derived from RRTEIObject

Public Attributes

Blue : Integer

Description

Specifies the amount of blue to use in the fill color for the RoseItemView object.

Green : Integer

Description

Specifies the amount of green to use in the fill color for the RoseItemView object.

Red : Integer

Description

Specifies the amount of red to use in the fill color for the RoseItemView object.

Transparent : Boolean

Description

Indicates whether the fill color of the RoseItemView object is transparent.

View_Font

Description

Specifies the amount of color, size, and style of the font to use in the for the ModelElementView object.

Derived from RRTEIObject

Public Attributes

Blue : Integer

Description

Specifies the amount of blue to use in the text color of a RoseItemView object.

Bold : Boolean

Description

Indicates whether the text's font style is Bold.

FaceName : String

Description

Specifies the text font name (such as Arial, Courier, etc.) of a RoseItemView object.

Green : Integer

Description

Specifies the amount of green to use in the text color of a RoseItemView object.

Italic : Boolean

Description

Indicates whether the text's font style is Italic.

Red : Integer

Description

Specifies the amount of red to use in the fill color for the RoseItemView object.

Size : Integer

Description

Specifies the text point size for a RoseItemView object.

StrikeThrough : Boolean

Description

Indicates whether the text's font style is Strikethrough.

Underline : Boolean

Description

Indicates whether the text's font style is Underline.

View_LineColor

Description

Specifies the amount of blue, green, or red to use in the line color for the ModelElementView object.

Derived from RRTElObject

Public Attributes

Blue : Integer

Description

Specifies the amount of blue to use in the line color for the ModelElementView object.

Green : Integer

Description

Specifies the amount of green to use in the line color for the ModelElementView object.

Red : Integer

Description

Specifies the amount of red to use in the line color for the ModelElementView object.

BasicScript Reference 4

Contents

- *Special Characters* on page 536
- *Directives* on page 573
- *Functions* on page 578
- *Keywords* on page 851
- *Methods* on page 857
- *Operators* on page 888
- *Properties* on page 913
- *Statements* on page 949
- *Optional Parameters* on page 1061
- *Arrays (topic)* on page 1198
- *Dialogs (topic)* on page 1219
- *Error Handling (topic)* on page 1220
- *Expression Evaluation (topic)* on page 1221
- *Keywords (topic)* on page 1223
- *Line Numbers (topic)* on page 1225
- *Literals (topic)* on page 1225
- *Named Parameters (topic)* on page 1227
- *Objects (topic)* on page 1228
- *Operator Precedence (topic)* on page 1231
- *Operator Precision (topic)* on page 1232
- *User-Defined Types (topic)* on page 1232

Special Characters

This chapter describes all of BasicScript reserved words available to you when creating your scripts.

' (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 result type is
Long	Single	Double
Boolean	Boolean	Integer

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.
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.

Constant	Value
SunOS	True if development environment is SunOS.
Ultrix	True if development environment is Ultrix.
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 result type 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 result type 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^01**
0^-nUndefined	0^+n0
1^n1	

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 line 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 result type is
Single	Long	Double

If one expression is	and the other expression is	then the result type is
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).

Data Types

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, handle = " & hWnd
          End If
End Sub
```

See Also

Declare (statement).

Platform(s)

All.

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 two-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 two bytes of storage.

When used within binary or random files, two 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)
- Double (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.

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)
- DefType (statement)
- CCur (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)
- DefType (statement)
- CDate, CVDate (functions)

Platform(s)

All.

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)
- DefType (statement)
- CDbl (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)
- DefType (statement)
- CInt (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                                         '0bject 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)
- DefType (statement)

Platform(s)

Windows, Win32, Macintosh.

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)

DefType (statement)

CSng (function)

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"
```

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)
- DefType (statement)
- CStr (function)

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, CurrencyIf 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)
- DefType (statement)
- CVar (function)
- VarType (function)

Platform(s)

All.

Directives

#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.
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.
Ultrix	True if development environment is Ultrix.
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.

Constant	Value
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.

Functions

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.

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.
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.

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.

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 returnsValue
Maximized	ebMinimized1
Minimized	ebMaximized2
Restored	ebRestored3

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
ProgMan
          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.

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.

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 FormatReturns
Asc	Value of the first byte of ***string*** (between 0 and 255) MBCSValue of the first character of ***string*** (between -32769 and 32767) WideValue of the first character of ***string*** after conversion to MBCS.

Function	String FormatReturns
AscB	Value of the first byte of ***string***. MBCSValue of the first byte of ***string***. WideValue of the first byte of ***string***.
AscW	Same as Asc. MBCSSame as Asc. WideValue 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
```

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)
- Sin (function)
- Cos (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.

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)
- Var (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.

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)

            'Displays the date passed as parameter 2.
            MsgBox "Item " & c% & " is '" & a & "'"
End Sub
```

See Also

- Switch (function)
- If (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 ***charcode***.

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	Value Between	Returns
Chr[$]	SBCS	0 and 255	A 1-byte character string.
	MBCS	-32768 and 32767	A 1-byte or 2-byte MBCS character string depending on ***charcode***.
	Wide	-32768 and 32767	A 2-byte character string.
ChrB[$]	SBCS	0 and 255	A 1-byte character string.
	MBCS	0 and 255	A 1-byte character string.
	Wide	0 and 255	A 1-byte character string.
ChrW[$]	SBCS	0 and 255	A 1-byte character string (same as the Chr and Chr$ functions)
	MBCS	-32768 and 32767	A 1-byte or 2-byte MBCS character string depending on ***charcode***.
	Wide	-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, AscB, AscW (functions)
- 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.

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

             'Displays 12367.
             MsgBox "The result is: " & CLng(i% * j&)

             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.

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.*A runtime error is generated if the specified combo box does not exist.
id	*Integer specifying the ID of the combo box.*

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.

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 msg
          Exit Sub                                  'Reset error
handler.
End Sub

'This 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

             'Create Visio object.
             Set Visio = CreateObject("visio.application")
             Set doc = Visio.Documents.Add("")
'Create a new doc.
             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.

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.

Platform Notes: 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]...]`

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, 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 TheDate$, then 'changes the
date and displays the result. It then changes 'the date back to the
saved date and displays the result.
Const crlf = Chr$(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.

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".

Named Parameter	Description
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***.
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 & " 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, 2000".
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 January 1 (i.e., the constant ebFirstJan1 as described bellow).

The following table lists the valid time interval strings and the meanings of each.

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 Format$ function uses the same expressions.

*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

Constant	Value	Description
ebFriday	6	Friday
ebSaturday	6	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).
ebFirstFourDays	2	The first week of the year is that containing at least four days in the year.
ebFirstFullWeek	3	The 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.

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 (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.

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)
- 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.

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 Nth 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 ñ1 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.

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 patternMatches these files	Doesn't match these files
S.TXT	SAMPLE.TXTGOOSE.TXTSAMS.TXT
	.SAMPLESAMPLE.DAT
C*T.TXT	CAT.TXT CAP.TXTA-
	CATS.TXT
C*T	CATCAP.TXT CAT.DOC
C?T	CATCUT CAT.TXTCAPITCT
*	(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 exam
```

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.

Platform Notes: UNIX

On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

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)

All.

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
End If
```

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.

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"
```

```
'set focus to another control
End If
DlgEnable "Files", False
'Now 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 (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).
'Does it have the focus?
        DlgFocus "OK"
'set focus to another control
End If
DlgEnable "Files", False
'Now 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.

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 items in list
box.
           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

- DlgControlId (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.

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, you 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 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. 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.

Action	Description
1	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.

Action	Description
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).
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 SampleDlgTemplate 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 SampleDlgTemplate
        SampleDialog.Printing = 1
        r% = Dialog(SampleDialog)
End Sub
```

See Also

- Begin Dialog (statement)

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.

Control Type	Value Returned by `DlgText$`
Text	Returns the label of the control.
Text box	Returns the label 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

```
'Display the text in the tenth control.
MsgBox DlgText$(10)

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	Control Type
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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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

Value	Meaning	Example
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: Win32, Windows, 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".

Platform Notes: UNIX

Under UNIX systems, the backslash and colon are valid filename characters.

Platform Notes: Macintosh

On the Macintosh, all characters are valid within filenames except colons, which are seen as path separators.

Platform Notes: 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.

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

Currently, 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.

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).
firstdayofweek	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).
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.

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.
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.

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.

Format	Description
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 list the characters you can use for numeric, string, and date/time formats and explain their functions.

Character	Meaning
Empty string	Displays the numeric expression as is, with no additional formatting.
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.

Character	Meaning
.	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.
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.

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.

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 (1ñ31).
dd	Displays the day with a leading 0 (01ñ31).
ddd	Displays the day of the week abbreviated (SunñSat).
dddd	Displays the day of the week (SundayñSaturday).
ddddd	Displays the date as a short date.
dddddd	Displays the date as a long date.
w	Displays the number of the day of the week (1ñ7). Sunday is 1; Saturday is 7.
ww	Displays the week of the year (1ñ53).

Character	Meaning
m	Displays the month without a leading 0 (1ñ12). If m immediately follows h or hh, m is treated as minutes (0ñ59).
mm	Displays the month with a leading 0 (01ñ12). If mm immediately follows h or hh, mm is treated as minutes with a leading 0 (00ñ59).
mmm	Displays the month abbreviated (JanñDec).
mmmm	Displays the month (JanuaryñDecember).
q	Displays the quarter of the year (1ñ4).
yy	Displays the year, not the century (00ñ99).
yyyy	Displays the year (1000ñ9999).
h	Displays the hour without a leading 0 (0ñ24).
hh	Displays the hour with a leading 0 (00ñ24).
n	Displays the minute without a leading 0 (0ñ59).
nn	Displays the minute with a leading 0 (00ñ59).
s	Displays the second without a leading 0 (0ñ59).
ss	Displays the second with a leading 0 (00ñ59).
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.
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.

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.
pmt	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.

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*

Platform Notes: 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
ebSystem	4	System files
ebVolume	9	Volume label

Constant	Value	Includes
ebDirectory	16	Subdirectories
ebArchive	32	Files that have changed since the last backup
ebNone	64	Files with no attributes

To determine 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 & _
           "Volume bit is set." &                      crlf
        If y% And ebDirectory Then message = message & _
```

```
            "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.

Platform Notes: 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.

Returned Value	Description
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 NT.

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.
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.

Note: 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.

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 NT.

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$	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.String containing the name of the combo box.
id	Integer specifying the ID of the combo box.

Note: 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.

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 NT.

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. 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.

Parameter	Description
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.

pathname	class	GetObject Returns
**	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.
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.

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.
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.

Platform Notes: 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")
```

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.

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.

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	Hanguel mode
Bit 3	Banja mode (single-byte)	Junga mode (double-byte)
Bit 4	Normal mode	Hanja conversation 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.

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.
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$ (functions)
- 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: 0String comparisons are case-sensitive. 1String 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.

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.

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.

Parameter	Description
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.

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.

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.

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.

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.

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.

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:

KillDir$ShellAppActivate

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.

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: s$ = “open chimes.wav type waveaudio” r% = Mci(s$,"")
error$	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 successful. 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
```

```
        'Wait for sound to finish.
        rc = Mci("play CoolSound wait","","")

        'Close driver and file.
        rc = Mci("close CoolSound","","")
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.

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$
- MidB
- MidB$ (statements)

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.

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., Jul., "
        mons$ = mons$ + "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.

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	1	Displays OK button only.
ebOKCancel	2	Displays OK and Cancel buttons.
ebAbortRetryIgnore	2	Displays Abort, Retry, and Ignore buttons.
ebYesNoCancel	3	DisplaysYes, 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	The current application is suspended until the dialog box is closed.
ebSystemModal	4096	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"
        MsgBox "This message box has Abort, Retry, and Ignore
buttons.", _
           ebAbortRetryIgnore,"MsgBox"
        MsgBox "This message box has Yes, No, and Cancel buttons.", _
           ebYesNoCancel Or ebDefaultButton2,"MsgBox"
        MsgBox "This message box has Yes and No
buttons.",ebYesNo,"MsgBox"
        MsgBox "This message box has Retry and Cancel buttons." , _
           ebRetryCancel,"MsgBox"
        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.

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.
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.
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.

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.

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.
hellofile	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"
```

Platform Notes: 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.

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.

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***.
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.

Named Parameter	Description
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.

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.

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.

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.

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.

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 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.
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
selection$	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.

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.

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.

RTrim, RTrim$ (functions)

Note: 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"
```

Platform Notes: 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"
```

Platform Notes: 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.

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
iInput	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.

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 ArrayOfElements. If the user selects Cancel, a value 1 less then 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.

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. ■ ebNormal: Focus: 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.

Note: 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: 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.

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.)

Platform Notes: 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""")
```

Platform Notes: 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:

- 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).
- BasicScript examines the PATH environment variable for an executable program called **sh** (the Bourne shell).
- 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.

Platform Notes: 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.

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.

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 parameter 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
	***Value* 12** - Returns a string containing the table qualifier used by the data source (e.g., "table," "file"). ***Value* 13** - Returns a string containing the database qualifier used by the data source (e.g., "database," "directory"). ***Value* 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.
qualifiertext	Optional String parameter required for actions 3, 4, or 5. The values are as follows: **Action 3 -** The ***qualifiertext*** parameter must be the name of the database represented by ID. **Action 4** - The ***qualifiertext*** parameter specifies a database name and an owner name. The syntax for this string is: ***DatabaseName.OwnerName*** **Action 5** - The ***qualifiertext*** parameter specifies the name of a table on the current connection.

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* 1** - Returns a one-dimensional array of available data sources. The array is returned in the ***resultarray*** parameter. ***Value* 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. ***Value* 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. ***Value* 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. ***Value* 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 ***Value* 6** - Returns a string containing the ID of the current user. ***Value* 7** - Returns a string containing the name (either the directory name or the database name, depending on the driver) of the current database. ***Value* 8** - Returns a string containing the name of the data source on the current connection. ***Value* 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”). ***Value* 10** - Returns a string containing the name of the server for the data source. ***Value* 11** - Returns a string containing the owner qualifier used by the data source (e.g., “owner,” “Authorization ID,” “Schema”).

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 1** - The driver's login dialog box is always displayed. **Value 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. **Value 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. **Value 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 1** - The driver's login dialog box is always displayed. **Value 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. **Value 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. **Value 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): (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.

Named Parameter	Description
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 parameter specifying a valid connection ID.

Named Parameter	Description
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.

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.

Parameter	Description
compare	Optional Integer specifying how the comparison is to be performed. It can be either of the following values: **Value 0** - Case-sensitive comparison **Value 1** - Case-insensitive comparison 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 a string to uppercase. This constant is supported on all platforms.
ebLowerCase	2	Converts a string to lowercase. This constant is supported on all platforms.
ebProperCase	3	Capitalizes the first letter of each word and lower-cases all 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.

Constant	Value	Description
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 INICODE 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, 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 parameter 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 256If 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.

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.

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.

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 msg
End Sub
```

See Also

- Time, Time$ (statements)
- Date, Date$ (functions)
- Date, Date$ (statements)
- Now (function)

Platform(s)

All.

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.

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.

Returned String	Returned if varname is
“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 particularly 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.

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.

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).

Value	Constant	Data Type
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.

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.

Named Parameter	Description
date	Any expression representing a valid date.

Named Parameter	Description
firstdayofweek	Indicates the first day of the week. If omitted, then Sunday is assumed (that is, the constant ebSunday described below).

The **Weekday** function takes the following named parameters:

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.

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.

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:

Named Parameter	Description
text$	String from which the sequence of words will be extracted.
firstInteger	Specifies the index of the first word in the sequence to return. If ***last*** is not specified, then only that word is returned.
lastInteger	Specifies 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.

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.

Keywords

_ (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 line 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.

. (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.

' (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.

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

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
          'The following displays 11 (Foo changed the value)
          MsgBox "The ByVal value is: " & i

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

See Also

ByRef (keyword)

Platform(s)

All.

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.

Methods

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 (that is, 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.

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.GetText (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

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"

        'Get list of bitmaps from Windows directory
        FileList list$,f$

        '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.

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	()
*Err.***Number**	()
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.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. Errors 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.

Named Parameter	Description
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.

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.
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.

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.

Platform Notes: Win32

The **Net.CancelCon** method 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."

        'This checks 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 Titles.
10	Articom Lantastic.
11	IBM AS/400.
12	FTP Software FTP NFS.
13	DEC Pathworks.

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.

Value Type 3 Returns the network driver version number.

Value Type 4 Returns 1 if the Net.User$ property is supported; returns 0 otherwise.

Value Type 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 Type 7 Returns a value indicating which printer function 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.

Value Type 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.

Platform Notes: Win32

For Win32 platforms, the ***type*** parameter can be any of the following values:

1 - Always returns 0.

2 - Network type:

Value of *type*	**Description**
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

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.

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.

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	Ctrl+PgDnScrolls 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.

Operators

& (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.

\ (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 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 result type 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 ***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.

> (operator)

See Comparison Operators (topic).

< (operator)

See Comparison Operators (topic).

<> (operator)

See Comparison Operators (topic).

- (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 result type is
Long	Single	Double
Boolean	Boolean	Integer

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.

* (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 result type is
Single	Long	Double
Boolean	Boolean	Integer

If one expression is	and the other expression is	then the result type is
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.

+ (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 result type 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.

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.

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
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.

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

If expression1 is	and expression2 is	then the result is
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.

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).

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 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.

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,SIX,SEVEN,EIGHT,"
        cval$ = cval$+"NINE,TEN,JACK,QUEEN,KING"
        Randomize
        card% = Random(1,52)
        value = card% Mod 13
        If value = 0 Then value = 13
        CardNum$ = 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.

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.

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.

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.

Properties

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.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:

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.

OS/2

Returns a string in the following format:

- ***countrycode***,***localename***
- ***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

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"
Win32	"Win32s", "Windows 95", "Windows NT"
OS/2	"OS/2"
Macintosh	"Macintosh"
Netware	"NetWare"
UNIX	"Lunix", "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 values returned from Basic.OperatingSystemVendor$
Windows	"Microsoft"
Win32	"Microsoft"
OS/2	"IBM"
Netware	Returns the name of the company that distributed NetWare.
Macintosh	"Apple"
UNIX	"Novell System Laboratories", "Lunix", "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.

Platform Notes: Windows, NetWare, OS/2

The version number is returns as ***major.minor***.

Platform Notes: 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 95Microsoft 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
6	ebU1trix	DEC Ultrix

Value	Constant	Platform
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.

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 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 "Invalid number, press ""F1"" to invoke help" _
                  ,,,Err.HelpFile,Err.HelpContext
            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 "Invalid number, press ""F1"" to invoke help" _
                 ,,, Err.HelpFile,Err.HelpContext
           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 and 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.

Platform Notes: OS/2

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.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.

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: Windows, Win32

Under Windows, this value is an **Integer**. Under Win32, this value is a **Long**.

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.

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.

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.

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.

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.

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.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.

Statements

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 (statement)

Platform(s)

Windows.

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.
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.

Platform Notes: 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.

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.

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"
            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.

Parameter	Description
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 Mgr.
          AppGetPosition x%,y%,0,0
'Retrieve its position.
          x% = x% + Screen.TwipsPerPixelX * 10
'Add 10 pixels.
          AppMove x% + 10,y%
'Nudge it 10 pixels
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 now 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.
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.

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 **Reassert** 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.

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.

Parameter	Description
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.
.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 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.

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.

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), FileList (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.

Platform Notes: Windows, Win32

BasicScript tracks and remembers the current directory for all drives in the system for that process.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: Macintosh

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

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.

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.

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:

- Must begin with a letter.
- May contain only letters, digits, and the underscore character.
- Must not exceed 80 characters in length.

- 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

DefType (statement), Let (statement), = (statement), Constants (topic)

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 TheDate$, then 'changes the
date and displays the result. It then changes 'the date back to the
saved date and displays the result.
Const crlf = Chr$(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

Platform	Minimum Date	Maximum Date
Windows 95	January 1, 1980	December 31, 2099
OS/2	January 1, 1980	December 31, 2079
NetWare	January 1, 1980	December 31, 2099

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.

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.
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 (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.

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 (statement), 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 (statement), 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.

Parameter	Description
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.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 IntegerDeclare _`
	`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.

Parameter	**Description**
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.

ParameterName	Name of the parameter, which must follow BasicScript naming conventions: - Must start with a letter. - 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. - 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-declaration 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.

Basic Script 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.

Supported Platform	Default Calling Conventions	Calling Convention
Windows	Pascal, CDecl	Pascal
Win32	Pascal, CDecl, StdCall	StdCall
Macintosh 68K	CDecl	CDecl
OS/2	System, Pascal, CDecl	System
NetWare	CDecl, Pascal	CDecl

Note: The Power Macintosh supports a single calling convention that evaluates parameters left to right. No special calling convention keywords are required. On the Power Macintosh, a runtime error occurs if any explicit calling convention keyword is specified.

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).

Data type	Is passed as
ByVal String	A pointer to a null-terminated string. With strings containing embedded nulls (Chr$(0)), it is not possible to determine which null represents the end of the string; therefore, the first null is considered the string terminator.An external routine can freely change the content of a string. It cannot, however, write beyond the end of the null terminator.
ByRef String	A pointer to a pointer to a null-terminated string. With strings containing embedded nulls (Chr$(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.
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):

- The current directory
- The Windows directory
- The Windows system directory
- The directory containing BasicScript
- 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):

4. The directory containing BasicScript
5. The current directory
6. The Windows system directory
7. The Windows directory
8. 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:

- If ***LibName$*** contains an explicit path, that code fragment will be loaded.
- If no path is specified in ***LibName$***, then BasicScript will look in the folder containing BasicScript, then the System folder.

- 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:

- Inside Macintosh: PowerPC System Software
- 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
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
```

Platform Notes: 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")
```

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.

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
Boolean	False

Data Type	Initial Value
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:

- Must start with a letter.
- 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.
- The last character of the name can be any of the following type-declaration characters: #, @, %, !, &, and $.
- Must not exceed 80 characters in length.
- 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

Public (statement), Private (statement), Option Base (statement)

Platform(s)

All.

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.

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)

All.

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

```
'Disable the Save Options control.
DlgEnable "SaveOptions", False

'Toggle a group of option buttons.
DlgEnable "EditingOptions"

'Enable six controls.
For i = 0 To 5
        DlgEnable i,True
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 (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"                                   'Set focus to
another control
End If
DlgEnable "Files", False                                'Now 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 (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 items in list box.
      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.

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.
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. 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.

Platform Notes: 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.

Platform Notes: 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

```
'Change text of group box 1.
DlgText "GroupBox1","Save Options"
If DlgText$(9) = "Save Options" Then
        'Change text to "Editing Options".
        DlgText 9,"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.

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 (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 OptionsTemplate 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 OptionsTemplate
        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 checks the condition first, then
'does the iteration 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.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes 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 (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.

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.

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.

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.
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**.

Element Type	What Erase Does to That Element
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.

        'Display element 1.
        MsgBox "Array before Erase: " & a$(1)
        Erase a$                    'Erase all elements in the array.

        'Display element 1 again (should be erased).
        MsgBox "Array after Erase: " & a$(1)
End Sub
```

See Also

Arrays (topic)

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.
HelpContex	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.

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 (statement), Sub...End Sub (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.

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.

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. 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.
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. TXTGOOSE.TXTSAMS.TXT	SAMPLESAMPLE.DAT
C*T.TXT	CAT.TXT	CAP.TXTACATS.TXT
C*T	CATCAP.TXT	CAT.DOC
C?T	CATCUT	CAT.TXTCAPITCT
*	(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

Constant	Value	Includes
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.

Platform Notes: UNIX

On UNIX platforms, the hidden file attribute corresponds to files without the read or write attributes.

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 that 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.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: 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 j = 1 To 10
Next j,i
```

```
For i = 1 To 10
                  For j = 1 To 10
                  Next j
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()
        For x = -1 To 0
           For y = -1 To 0
              Z = x Or y
              message = message & Format(Abs(x%),"0") & " Or "
              message = message & Format(Abs(y%),"0") & " = "
              message = message & Format(Z,"True/False") & Basic.Eoln$
           Next y
        Next x
        MsgBox message
End Sub
```

See Also

Do...Loop (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 Win32, you can break out of infinite loops using Ctrl+Break.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: 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.

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.
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.

Part	Description
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`
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:

- 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:

  'Only passes two out of three required parameters.

  a = Test(,1)

  'Only passes two out of three required parameters.

  a = Test(1,2)

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)
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.

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.

Parameter	Description
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.
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#,,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).

Variable Type	File Storage Description
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.
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.

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:

- Must begin with a letter.
- May contain letters, digits, and the underscore character.
- Must not exceed 80 characters in length.
- 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.

Platform Notes: UNIX

To break out of an infinite loop, press Ctrl+C.

Platform Notes: Macintosh

To break out of an infinite loop, press Ctrl+Period.

Platform Notes: 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 if 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.

Parameter	Description
.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 HelpDialogTemplate
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.

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.

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.

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.

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.

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.

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.

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:

```
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.

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.

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.

Parameter	Description
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.

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).

Platform Notes: UNIX

Under all versions of UNIX, file locking is ignored.

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$(40, "*")
        message = "Here are two strings that have been " & crlf
        message = message & "right- and left-justified in a" & _
           " 40-character string." & crlf & crlf
        RSet tmpstr$ = "Right->"
        message = message & tmpstr$ & crlf
        LSet tmpstr$ = "<-Left"
        message = message & tmpstr$ & crlf
        MsgBox message
End Sub
```

See Also

RSet (statement)

Platform(s)

All.

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.

Platform Notes: Macintosh

Requires Macintosh System 7.0 or later.

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.

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$, MidB, MidB$ (functions), Option Compare (statement)

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.

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"
```

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"
Kill "c:\samples\mydoc.txt"
```

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 can't 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 can't be 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.

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.

Parameter	Description
.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.

File Mode	Description
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.
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

File Mode	Default Value for *accessmode*
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 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.

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)
\?	Question mark	?
\'	Single quote	'
\x***hh***	Hexadecimal number	Chr$(Val(&H***hh***))
ooo	Octal number	Chr$(Val(&O***ooo***))

Escape	Description	Equivalent Expression
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.
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.

Platform Notes: Macintosh

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

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.

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.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: 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.

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.

Data Type	Description
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.

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
```

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.

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
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.

Platform Notes: 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).

VariableType	File Storage Description
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 valueand 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.
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 msg
         Close
         Kill "test.dat"
End Sub
```

See Also

Open (statement), Put (statement), Write# (statement), Print# (statement)

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$*** is described under the SendKeys statement.
time	Integer specifying the number of milliseconds devoted for the output of the entire ***KeyString$*** 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.

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 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  'Double-click,
        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), 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.
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.

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.

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. 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.

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.

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 msg,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.

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.

Platform Notes: 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")
```

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.

Part	Description
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.

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.

Named Parameter	Description
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.

Platform Notes: 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.

Parameter	Description
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.
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.

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.

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.

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.
Static	Recognized by the compiler but currently has no effect.
name-	Name of the subroutine, which must follow BasicScript naming conventions: - Must start with a letter. - 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. - Must not exceed 80 characters in length.

Part	Description
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.

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.
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 fon. 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.

Platform Notes: OS/2

Under OS/2, accelerators are underlined, and the Alt+***letter*** accelerator combination is used.

Platform Notes: 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: ■ ebRegularNormal font (i.e., neither bold nor italic) ■ ebBoldBold fontebItalicItalic font ■ ebBoldItalicBold-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$ (statements)

Syntax

```
Time[$] = newtime
```

Description

Sets the system time to the time contained in the specified string.

Comments

The **Time$** statement requires 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$
        msg = "Time was: " & oldtime$ & crlf
        Time$ = "10:30:54"
        msg = msg & "Time set to: " & Time$ & crlf
        Time$ = oldtime$
        msg = msg & "Time restored to: " & Time$
        MsgBox msg
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.

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.

Unlock (statement)

See **Lock**, **Unlock** (statements).

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.

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
        Message$="We will now go to the the top..."
        MsgBox Message$
        VScroll 0
        VScroll 0
        MsgBox "...and here we are!"
        Viewport.Close
End Sub
```

See Also

VLine (statement), VPage (statement)

Platform(s)

Windows.

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 inadvertently create infinite loops in your code. Under Windows and Win32, you can break out of infinite loops using Ctrl+Break.

Platform Notes: 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.

Platform Notes: 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.

Platform Notes: 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.

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".

Parameter	Description
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, the current window can be an MDI child window, a pop-up window, or a top-level window.

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.

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.

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.

Parameter	Description
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.

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.
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.

Use the command	To
Erase	Erase all the elements of an array.
ReDim	Establish the bounds and dimensions of an array.
Dim	Declare an array.

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.

Constants (topic)

Constants are variables that cannot change value during script execution. The following constants are predefined by BasicScript.

Constant	Value	Description
ebMinimized	1	The application is minimized.
ebMaximized	2	The application is maximized.
ebRestored	3	The application is restored.

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.

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$(0)	String containing a single null character.
ebNullString	0	Special string value used to pass null pointers to external routines.
ebTab	Chr$(9)	String containing a tab.
ebVerticalTab	Chr$(12)	String containing a vertical tab.

Constant	Value	Description
ebCFText	1	Text.
ebCFBitmap	2	Bitmap.
ebCFMetafile	3	Metafile.
ebCFDIB	8	Device-independent bitmap.
ebCFPalette	9	Palette.
ebCFUnicode	13	Unicode text.

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 (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.
Ultrix	True if development environment is Ultrix.
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

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.
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.

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.

Constant	Value	Description
ebDOS	1	A DOS executable file.

Constant	Value	Description
ebWindows	2	A Windows executable file.

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.

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.

Constant	Value	Description
PI	3.1415...	Value of PI.

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.
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.

Constant	Value	Description
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.

Constant	Value	Description
ebWin16	0	Microsoft Windows (16-bit).
ebWin32	2	Microsoft Windows 95Microsoft Windows NT WorkstationMicrosoft Windows NT ServerMicrosoft Win32s running under Windows 3.1
ebSolaris	3	Sun Solaris 2.x
ebSunOS	4	SunOS
ebHPUX	5	HP-UX
ebUltrix	6	DEC Ultrix
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

Constant	Value	Description
ebLandscape	1	Landscape paper orientation.
ebPortrait	2	Portrait paper orientation.

Constant	Value	Description
ebLeftButton	1	Left mouse button.
ebRightButton	2	Right mouse button.

Constant	Value	Description
ebHide	0	Application is initially hidden.
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	5	Application is minimized and does not have the focus.

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.

Constant	Value	Description
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.

Constant	Value	Description
ebEmpty	0	Variant has not been initialized.
ebNull	1	Variant contains no valid data.
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.**

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:

- 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.
- 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.

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:

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:

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.

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.

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
```

Keywords (topic)

A keyword is any word or symbol recognized by BasicScript as part of the language. All of the following are keywords:

Operator	Description	Precedence Order
()	Parentheses	Highest
^	Exponentiation	
-	Unary minus	
/, *	Division and multiplication	
\	Integer division	

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	Re	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 listed above 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.

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
```

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

Literal	Description
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."
```

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!!!
```

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.

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	
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.

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.

Index

. (keyword) 547, 852
(operator) 555, 892
- (operator) 537, 892
/ (operator) 548, 890

Symbols

554, 555, 892
(operator) 550, 889
#Const (directive) 539, 573
#If...Then...#Else (directive) 540, 574
& (operator) 543, 888
' (keyword) 536, 853
( ) (keyword) 544
* (operator) 546, 894
+ (operator) 553, 895
= (operator) 556
= (statement) 555
> (operator) 556, 892
>= (operator) 556
^ (operator) 550, 891
_ (keyword) 551, 851

A

A Polling Add-In (automation) 15
About Collection Attributes and Operations 4
About Default Properties and Property Sets (Extensibility) 4
Abs (function) 578
Accessing Object Methods 1229
Accessing Object Properties 1228
Action 396
Action Classes 393
ActionMode 398
ActivateControl (statement) 949
AddIn 86
adding
Watch Variables (Extensibility Interface) 44
Adding a Comment at the End of a Line of Code 39
Adding a Full-Line Comment 38
Adding a Property to a Set 23
Adding Comments to a Script 38
Adding Controls 57
Adding Entries to a Rational Rose RealTime Menu File 8
Adding Pictures to a Dialog 59
Adding Scripts to a Rational Rose RealTime Menu 13
AddInManager 93
Add-only 338
Alignment 1216
And (operator) 897
AnswerBox (function) 579
Any (data type) 556
AppActivate (statement) 951
AppClose (statement) 953
AppFileName$ (function) 582
AppFind, AppFind$ (functions) 583
AppGetActive$ (function) 584
AppGetPosition (statement) 954
AppGetState (function) 585
AppHide (statement) 956
Application 93
Application Classes 81
AppList (statement) 957
AppMaximize (statement) 958
AppMinimize (statement) 960
AppMove (statement) 961
AppRestore (statement) 963
AppSetState (statement) 964
AppShow (statement) 966
AppSize (statement) 967
AppType (function) 587
ArrayDims (function) 589
Arrays (topic) 1198
ArraySort (statement) 969
Asc, AscB, AscW (functions) 590

AskBox, AskBox$ (functions) 591
AskPassword, AskPassword$ (functions) 593
Assigning a Value to an Object Variable 1228
Association 290
Association Classes 288
AssociationEnd 294
AssociationEndContainment 298
AssociationEndRole 350
AssociationEndVisibilityKind 299
AssociationRole 351
Atn (function) 595
Attribute 338
attribute 338
  AttributeContainment 338
  changeable 338
  GetChangeable 338
  OwnerScope 339
attribute (RRTEI) 338
AttributeContainment 340
AttributeVisibilityKind 339, 340
Automation 3
automation
  Extensibility Interface 3

B

Basic.Architecture$ (property) 913
Basic.Capability (method) 857
Basic.CodePage (property) 914
Basic.Eoln$ (property) 915
Basic.FreeMemory (property) 916
Basic.HomeDir$ (property) 917
Basic.Locale$ (property) 918
Basic.OperatingSystem$ (property) 920
Basic.OperatingSystemVendor$ (property) 921
Basic.OperatingSystemVersion$ (property) 923
Basic.OS (property) 924
Basic.PathSeparator$ (property) 926
Basic.Processor$ (property) 926
Basic.ProcessorCount (property) 928
BasicScript Reference 535
Basic.Version$ (property) 928
Beep (statement) 970
Begin Dialog (statement) 971
BranchPointView 519
breakpoints
  setting 43
ButtonEnabled (function) 595
ButtonExists (function) 597
ByRef (keyword) 854
Byte Ordering with Files 1215
Byte Ordering with Structures 1215
ByVal (keyword) 855

C

Call (statement) 974
CallAction 399
Calling Conventions with External Routines 999
Calling External Routines in Multi-Threaded Environments 1004
CancelButton (statement) 975
Capsule 303
CapsuleRole 352
CapsuleRoleView 494
CapsuleStructure 353
CapsuleView 477
Capturing Standard Windows dialogs 51
Cascading Errors 1220
CBool (function) 598
CCur (function) 599
CDate, CVDate (functions) 600
CDbl (function) 602
Changeable 338
Changing Titles and Labels 51
ChDir (statement) 977
ChDrive (statement) 978
CheckBox (statement) 979
CheckBoxEnabled (function) 603
CheckBoxExists (function) 604
ChoicePoint 418
ChoicePointView 519
Choose (function) 605
Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ (functions) 606
CInt (function) 608
Class 304
Class Diagram Classes 475

ClassConcurrency 310
ClassDependency 433
ClassDiagram 477
Classifier 310
Classifier Classes 299
ClassifierRole 356
ClassifierRoleView 514
ClassifierView 490
ClassifierVisibilityKind 327
ClassKind 310
ClassRelation 434
ClassView 490
Clipboard$ (function) 610
Clipboard$ (statement) 981
Clipboard.Clear (method) 859
Clipboard.GetFormat (method) 859
Clipboard.GetText (method) 861
Clipboard.SetText (method) 862
CLng (function) 611
Close (statement) 982
Collaboration 358
Collaboration Classes 347
Collaboration Diagram Classes 493
CollaborationDiagram 496
Collection 131
Collections 1230
ComboBox (statement) 983
ComboBoxEnabled (function) 612
ComboBoxExists (function) 614
Command, Command$ (functions) 615
Comments (topic) 1201
Common Logical View Enumerations 372
Comparing Object Variables 1229
Comparison Operators (topic) 1201
Compiling Your Script 47
Component 149
component
    setting topc capsule using RRTEI 69
Component Diagram Classes 501
Component View Classes 145
ComponentDependency 435
ComponentDiagram 502
ComponentInstance 249
ComponentPackage 170
ComponentPackageView 508
ComponentView 509
CompositeState 419
CompositeStateView 521
Connector 364
Const (statement) 985
Constant Folding 1226
Constants (topic) 1205
contacting Rational customer support xvii
ContextMenuItem 122
Continuing Debugging at a Line Outside the Current Subroutine 43
ControllableElement 184
Copying a Selection 38
Copying Structures 1233
Core Model Classes 178
Coregion 399
CoregionView 522
Cos (function) 616
CreateAction 401
CreateMessageView 514
CreateObject (function) 617
Creating a New Property 20
Creating a New Property Set 20
Creating a New Rational Rose RealTime Menu File 9
Creating a New Script from an Existing Script 13
Creating New Objects 1016
Creating New Rational Rose RealTime Scripts 13
Cross-Platform Scripting (topic) 1214
CSng (function) 619
CStr (function) 620
CurDir, CurDir$ (functions) 621
Currency (data type) 559
customizing
    Rational Rose RealTime Menus 7
Customizing Rational Rose RealTime Menus 7
Cutting a Selection 38
CVar (function) 622
CVErr (function) 624

D

Data Types 556
Date (data type) 560

Date, Date$ (functions) 625
Date, Date$ (statements) 987
DateAdd (function) 626
DateDiff (function) 628
DatePart (function) 631
DateSerial (function) 634
DateValue (function) 635
Day (function) 636
DDB (function) 637
DDEExecute (statement) 989
DDEInitiate (function) 638
DDERequest, DDERequest$ (functions) 640
DDESend (statement) 990
DDETerminate (statement) 991
DDETerminateAll (statement) 993
DDETimeout (statement) 994
Debugging Selected Portions of Your Script 43
Declare (statement) 995
Declaring Array Variables 1198
Declaring Explicit OLE Automation Objects 1016
Declaring Object Variables 1228
Declaring Structures 1232
Default Properties 1222
DefaultModelProperties 194
DefType (statement) 1009
DeleteSetting (statement) 1012
Deleting a Model Property 22
Deleting Model Properties 20
Deleting Text 38
Deleting Watch Variables 46
Deleting, Cutting, Copying, and Pasting Text 38
Deployment Diagram Classes 509
Deployment View Classes 246
DeploymentDiagram 510
DeploymentPackage 252
Description 86
Desktop.ArrangeIcons (method) 863
Desktop.Cascade (method) 864
Desktop.SetColors (method) 865
Desktop.SetWallpaper (method) 866
Desktop.Snapshot 867
Desktop.Tile (method) 869
DestroyAction 401
Device 258
Diagram 454
Dialog (function) 641
Dialog (statement) 1013
Dialogs (topic) 1219
Dim (statement) 1014
Dir, Dir$ (functions) 644
Directives 573
DiskDrives (statement) 1018
DiskFree (function) 647
Displaying and Adjusting the Grid 49
Displaying the Calls dialog 42
Displaying the Information Dialogs 60
DlgCaption (function) 648
DlgCaption (statement) 1019
DlgControlId (function) 648
DlgEnable (function) 650
DlgEnable (statement) 1020
DlgFocus (function) 651, 652
DlgFocus (statement) 1021
DlgListBoxArray (function) 654
DlgListBoxArray (statement) 1022
DlgProc (function) 655
DlgSetPicture (statement) 1024
DlgText (statement) 1026
DlgText$ (function) 659
DlgValue (function) 661
DlgValue (statement) 1027
DlgVisible (function) 662
DlgVisible (statement) 1029
DoEvents (function) 664
DoEvents (statement) 1034
DoKeys (statement) 1036
Do...Loop (statement) 1032
Double (data type) 561
Drive Letters 1218
DropListBox (statement) 1037
Duplicating Controls 58
Dynamic Arrays 1199

E

EditEnabled (function) 664
EditExists (function) 666

editing
Virtual Path for Scripts 15
Editing an Existing dialog 49
Element 204
End (statement) 1039
Environ, Environ$ (functions) 667
Environment 376
EOF (function) 668
Eqv (operator) 899
Erase (statement) 1040
Erl (function) 669
Err.Clear (method) 869
Err.Description (property) 929
Err.HelpContext (property) 930
Err.HelpFile (property) 932
Err.LastDLLError (property) 934
Err.Number (property) 936
Error (statement) 1041
Error Handlers 1220
Error Handling (topic) 1220
Error, Error$ (functions) 670
Err.Raise (method) 871
Err.Source (property) 937
Event 409
Event Classes 407
EventGuard 409
Example 20, 21, 22, 23, 24, 26, 28, 30, 31, 803
Exit Do (statement) 1043
Exit For (statement) 1044
Exit Function (statement) 1045
Exit Sub (statement) 1046
Exp (function) 671
Expression Evaluation (topic) 1221
Extensibility Classes 130
Extensibility Interface 2
Accessing Collection Elements By Count 30
Accessing Collection Elements By Name 30
Accessing Collection Elements By Unique ID 31
Adding a Property to a Set 23
Adding Entries to a Rational Rose RealTime Menu File 8
Adding or Editing the Virtual Path for Scripts 15
Adding Scripts to a Rational Rose RealTime Menu 13
automation 3
Cloning a Property Set 25
Collection Attributes 4
Collection Property 5
compiling script 47
Creating a New Property 20
Creating a New Property Set 20
Creating a New Rational Rose RealTime Menu File 9
Creating a New Script from an Existing Script 13, 34
Creating a New Script from Scratch 34
Creating a New Tool 29
Creating a User-Defined Property Type 24
Creating New Rational Rose RealTime Scripts 13, 34
Default Properties 4
Deleting a Model Property 22
Deleting Model Properties 20
Displaying the Calls dialog 42
Getting an Element from a Collection (Overview) 30
Getting and Setting the Current Property Set 21
Getting Model Properties 22
Managing Default Properties 18
Menu Extensibility 6
Methods for All Collections 5
Methods for User-defined Collections 6
Model Properties 17
Opening a Script 34
Operations 4
Placing Classes in LogicalPackages 32
Property Sets 4
Running, Pausing, and Stopping Your Script 41
Scripting 3
Setting and Removing Breakpoints 43
Setting Model Properties 27
Setting Model Properties Using InheritProperty 27
Setting Model Properties Using OverrideProperty 28

Tracing Script Execution 41
type libraries 3
Using Rational Rose RealTime Automation 14, 32
Using Rational Rose RealTime Script 14, 32
Watch Variables 44
Working with Classes 32
Working with Collections 29
Working with Rose RealTime Automation 32
ExternalDocument 215

F

Feature Classes 336
File Attributes 1051
FileAttr (function) 672
FileCopy (statement) 1047
FileDateTime (function) 674
FileDirs (statement) 1048
FileExists (function) 675
FileLen (function) 676
FileList (statement) 1050
FileParse$ (function) 677
FileType (function) 679
FinalState 424
FinalStateView 522
Finding and Replacing Text 39
Finding Specified Text 39
Fix (function) 680
Fixed Arrays 1199
Fixed-Length Strings 1015
For Each...Next (statement) 1053
Format, Format$ (functions) 681
For...Next (statement) 1055
FreeFile (function) 690
Frozen 338
Function...End Function (statement) 1058
Functions 578
Fv (function) 691

G

Generalization 436
GeneralizationVisibilityKind 438
Genericity 367
Get (statement) 1063
GetAllSettings (function) 692
GetAttr (function) 694
GetChangeable 338
changeable 338
frozen 338
GetCheckBox (function) 696
GetComboBoxItem$ (function) 698
GetComboBoxItemCount (function) 699
GetEditText$ (function) 701
GetListBoxItem$ (function) 702
GetListBoxItemCount (function) 704
GetObject (function) 705
GetOption (function) 706
GetSetting (function) 708
Getting and Setting the Current Property Set 21
Getting Model Properties 22
Getting the Rational Rose RealTime Application Object 14
GetToolNames 202
Global (statement) 1066
GoSub (statement) 1066
Goto (statement) 1068
GroupBox (statement) 1069
Guidelines for Using a Script to Call Another Script 48

H

HelpButton (statement) 1071
Hex, Hex$ (functions) 709
HLine (statement) 1072
Hour (function) 710
How To 20, 21, 22, 25, 27, 28, 30, 31
HPage (statement) 1073
HScroll (statement) 1074
HWND.Value (property) 939

I

If...Then...Else (statement) 1075
IIf (function) 711
IMEStatus (function) 712

Imp (operator) 901
Implicit Variable Declaration 1015
Incorporating dialogs or Controls into Your Script 53
InitialPoint 425
InitialPointView 523
Inline (statement) 1077
Input, Input$, InputB, InputB$ (functions) 715
InputBox, InputBox$ (functions) 716
Inserting a dialog into Your Script 48
InstantiateRelation 438
InStr, InStrB (functions) 718
Int (function) 721
Integer (data type) 563
Interaction 376
Interaction Classes 374
InteractionInstance 382
InteractionInstanceView 515
IPmt (function) 721
IRR (function) 723
Is (operator) 903
IsDate (function) 725
IsEmpty (function) 726
IsError (function) 727
IsMissing (function) 729
IsNull (function) 730
IsNumeric (function) 731
IsObject (function) 732
Item$ (function) 733
ItemCount (function) 735

J

JunctionAdornmentView 523
JunctionContinuationMode 425
JunctionPoint 425
JunctionPointView 524

K

Keywords 851
Keywords (topic) 1223
Kill (statement) 1078

L

LBound (function) 736
LCase, LCase$ (functions) 737
Left, Left$, LeftB, LeftB$ (functions) 738
Len, LenB (functions) 739
Let (statement) 1080
LifeLineView 515
Like (operator) 905
Line Numbers (topic) 1225
Line$ (function) 742
LineCount (function) 743
LineVertex 529
ListBox (statement) 1081
ListBoxEnabled (function) 744
ListBoxExists (function) 746
Literals (topic) 1225
Loc (function) 747
LocalState 402
LocalStateOrActionView 525
Lock, Unlock (statements) 1083
Lof (function) 748
Log (function) 749
Logical Package Structure 80
Logical View Classes 267
LogicalPackage 269
LogicalPackageDependency 439
LSet (statement) 1086
LTrim, LTrim$ (functions) 750

M

MacID (function) 750
MacScript (statement) 1087
Main (statement) 1088
Managing Default Properties (Extensibility) 18
Mci (function) 751
menu
  actions 12
  adding scripts 13
Menu Actions 12
menu extensibility
  Extensibility Interface 6
Menu File
  adding entries (Extensibility Interface) 8

menu file
creating new (Extensibility Interface) 9
keywords 11
sample 9
Menu File Keywords 11
menu files
syntax rules 10
MenuItemChecked (function) 754
MenuItemEnabled (function) 755
MenuItemExists (function) 756
menus
customizing (Extensibility Interface) 7
MenuState 123
Message 385
MessageEnd 386
MessageView 516
Methods 857
Mid, Mid$, MidB, MidB$ (functions) 757
Mid, Mid$, MidB, MidB$ (statements) 1089
Minute (function) 758
MIRR (function) 759
MkDir (statement) 1090
Mod (operator) 907
Model 218
model
opeing using RRTEI 66
Model Classes 145
ModelElement 236
Modifiers 11
modifying
property value using RRTEI 67
Month (function) 761
Moving the Insertion Point in a Script 34
Moving the Insertion Point to a Specified Line in Your Script 35
Moving the Insertion Point with the Mouse 35
MsgBox (function) 762
MsgBox (statement) 1091
Msg.Close (method) 873
Msg.Open (method) 874
Msg.Thermometer (property) 940

N

Name (statement) 1092
Named Parameters (topic) 1227
Net.CancelCon (method) 875
Net.Dialog (method) 877
Net.GetCaps (method) 877
Net.GetCon$ (method) 883
Net.User$ (method) 884
New (keyword) 856
Not (operator) 908
NoteView 464
Now (function) 766
NPer (function) 767
Npv (function) 768

O

Object (data type) 564
Objects (topic) 1228
Oct, Oct$ (functions) 770
OKButton (statement) 1093
On Error (statement) 1095
Open (statement) 1097
OpenFileName$ (function) 771
opening
maodel using extensibility interface 66
model using RRTEI 66
Operation 340
OperationConcurrency 345
Operations on Arrays 1200
OperationVisibilityKind 345
Operator Precedence (topic) 1231
Operator Precision (topic) 1232
Operators 888
Option Base (statement) 1100
Option Compare (statement) 1101
Option CStrings (statement) 1103
Option Default (statement) 1104
Option Explicit (statement) 1105
Optional Parameters 1061
OptionButton (statement) 1106
OptionEnabled (function) 773
OptionExists (function) 774
OptionGroup (statement) 1107

Or (operator) 909
OverrideProperty 67
OwnerScope 339, 346

P

Package 239
Parameter 328
ParentClassifier 339
Passing Arrays 1200
Passing Data to External Routines 1001
Passing Null Pointers 1001
Passing Parameters 998
Passing Parameters to Functions 1060
Passing Structures 1233
Pasting Items into Dialog Editor 60
Pasting the Contents of the Clipboard into Your Script 38
Path Separators 1217
PathMap 124
pausing
  Executing Script 41
Pausing an Executing Script 41
Picture (statement) 1108
Picture Caching 1030
PictureButton (statement) 1111
Platform(s) 1225
Pmt (function) 775
PopupMenu (function) 777
Port 367
Portability of Compiled Code 1216
PortEvent 411
PortRole 369
PortRoleView 499
PortView 500
PortVisibilityKind 370
PPmt (function) 778
Predefined Objects 1230
Print (statement) 1114
PrinterGetOrientation (function) 780
PrinterSetOrientation (statement) 1116
PrintFile (function) 781
Private (statement) 1117
Processor 262
Properties 913
Property 243
property value
  modifying using RRTEI 67
Protocol 329
ProtocolRoleEvent 415
ProtocolView 492
Public (statement) 1119
Public Attributes 86, 387
Public Operations 88
PushButton (statement) 1121
Put (statement) 1123
Pv (function) 782

Q

QueEmpty (statement) 1126
QueFlush (statement) 1127
QueKeyDn (statement) 1128
QueKeys (statement) 1129
QueKeyUp (statement) 1131
QueMouseClick (statement) 1132
QueMouseDblClk (statement) 1133
QueMouseDblDn (statement) 1134
QueMouseDn (statement) 1135
QueMouseMove (statement) 1136
QueMouseMoveBatch (statement) 1137
QueMouseUp (statement) 1139
Querying Arrays 1200
Querying the Capabilities of a Platform 1214
Querying the Platform 1214
QueSetRelativeWindow (statement) 1140

R

Random (function) 783
Randomize (statement) 1141
Rate (function) 784
Rational customer support
  contacting xvii
Rational Rose RealTime Menu Extensibility 6
Reading and Writing to Text Files 1216
ReadIni$ (function) 786
ReadIniSection (statement) 1142

RealizeRelation 440
ReDim (statement) 1143
RegistrationMode 370
Relation 442
Relation Classes 431
Relative Paths 1218
Rem (statement) 1145
Removing a Single Breakpoint Manually 44
Removing All Breakpoints Manually 44
Replacing Specified Text 40
ReplyAction 402
Repositioning Items 55
RequestAction 402
Reset (statement) 1145
Resizing Items 56
ResponseAction 403
Restrictions 1224
Resume (statement) 1146
Return (statement) 1148
ReturnAction 404
Returning Values from External Routines 1003
Returning Values from Functions 1060
RichType 142
RichTypes 141
RichTypeValuesCollection 144
Right, Right$, RightB, RightB$ (functions) 787
RmDir (statement) 1149
Rnd (function) 788
RoseBase 139
RoseRTApp.CurrentModel Example (Automation) 15
RoseRTApp.CurrentModel Example (Scripting) 14
Rosescript 12
Rounding 1222
RREEI
  attribute 338
RRTEI
  modifying a property value 67
  opening a model 66
  setting the top capsule
    capsule
      setting a TOP Capsule using RRTEI 69
RRTEI - see Extensibility Interface 1
RRTEI Model 2
RRTEIObject 140
RsActionKind 387
RsActionMode 404
RsClassKind 332
RsConcurrency 334
RsContainment 372
RSet (statement) 1150
RsExternalDocumentType 244
RsGenericity 370
RsJunctionContinuationMode 427
RsMenuState 127
RsNoteViewType 466
RsOwnerScope 346
RsRegistrationMode 371
RsSendActionPriority 405
RsSourceRegionType 388
RsStateKind 427
RsStereotypeDisplay 466
RsVisibilityKind 373
RTrim, RTrim$ (functions) 789
running
  Script 41
Running Your Script 41
Running, Pausing, and Stopping Your Script 41

S

Sample Rational RoseRT Menu File 9
SaveFileName$ (function) 789
SaveSetting (statement) 1151
scaript
  deleting watch variables 46
Screen.DlgBaseUnitsX (property) 941
Screen.DlgBaseUnitsY (property) 942
Screen.Height (property) 943
Screen.TwipsPerPixelX (property) 943
Screen.TwipsPerPixelY (property) 944
Screen.Width (property) 945
Script
  Stepping Through Your Script 41
script
  Assigning Accelerator Keys 51
  Attributes You Can Adjust 63

compiling 47
Debugging Interscript Calls 48
incorporating dialogs and controls 53
Interscript Calls 48
Modifying the Value of Variables on the Watch Variable List 46
pausing 41
running 41
setting breakpoints 43
stopping 41
watch variables 44, 46
Scripting 3
scripting
Extensibility Interface 3
RRTEI 3
scripts
adding to a menu 13
Second (function) 792
See Also 793
Seek (function) 793
Seek (statement) 1153
SelectBox (function) 794
SelectButton (statement) 1156
Select...Case (statement) 1154
SelectComboBoxItem (statement) 1157
Selecting an Entire Line 37
Selecting Controls 54
Selecting dialogs 54
Selecting Text 36
Selecting Text with the Keyboard 37
Selecting Text with the Mouse 36
Selecting Variables on the Watch List 46
SelectListBoxItem (statement) 1159
SendAction 406
SendActionPriority 407
SendKeys (statement) 1160
Sequence Diagram Classes 513
SequenceDiagram 516
Set (statement) 1164
SetAttr (statement) 1165
SetCheckBox (statement) 1167
SetEditText (statement) 1168
SetOption (statement) 1169
Setting Model Properties 27
Sgn (function) 796
Shell (function) 797
Signal 335
Sin (function) 800
Single (data type) 565
Size of Structures 1234
Sleep (statement) 1170
Sln (function) 801
SourceRegionType 389
Spc (function) 802
Special Characters 535
Specifying a Virtual Path for Scripts 15
SQLBind (function) 803
SQLClose (function) 805
SQLError (function) 806
SQLExecQuery (function) 808
SQLGetSchema (function) 809
SQLOpen (function) 812
SQLRequest (function) 814
SQLRetrieve (function) 817
SQLRetrieveToFile (function) 819
Sqr (function) 821
Starting Debugging Partway through a Script 43
State Classes 416
State Diagram Classes 517
State Machine Classes 387
StateDiagram 525
StateKind 428
StateMachine 389
Statements 949
StatePerimeterView 527
StateVertex 429
StereotypeDisplay 467
Stop (statement) 1171
stopping
Executing Script 41
Stopping an Executing Script 41
Str, Str$ (functions) 822
StrComp (function) 823
StrConv (function) 825
String (data type) 567
String, String$ (functions) 827
StructuredProperty 244
StructurePerimeterView 500
Sub...End Sub (statement) 1172
Switch (function) 828

SYD (function) 829
Syntax Rules for Rational Rose RealTime Menu File Entries 10
System.FreeMemory (property) 945
System.FreeResources (property) 946
System.TotalMemory (property) 947
System.WindowsDirectory$ (property) 948
System.WindowsVersion$ (property) 948

T

Tab (function) 831
Tan (function) 832
TerminateAction 407
Testing Your dialogs 52
Text (statement) 1175
TextBox (statement) 1177
The RRTEI Model and Rational Rose RealTime Extensibility 2
The Script Editor Window 33
Time, Time$ (functions) 833
Time, Time$ (statements) 1180
Timer (function) 834
TimeSerial (function) 834
TimeValue (function) 835
top capsule
  setting using RRTEI 69
  setting using the extensibility interface 69
Tracing Script Execution 41
Transition 390
Trim, Trim$, LTrim, LTrim$, RTrim, RTrim$ (functions) 836
Type (statement) 1181
Type Coercion 1221
Type Libraries 3
  Extensibility Interface 3
TypeName (function) 838
TypeOf (function) 840
TypeSafeSignals 67

U

UBound (function) 840
UCase, UCase$ (functions) 842
UNC Pathnames 1219
UninterpretedAction 407
Unlock (statement) 1182
Unsupported Language Elements 1217
Use Case View Classes 444
UseCase 445
User-Defined Types (topic) 1232
UsesRelationVisibilityKind 444
Using Interscript Calls 48

V

Val (function) 843
Variable Types 1064
Variant (data type) 569
VarType (function) 844
View Classes 450
View Property Classes 528
View_FillColor 530
View_Font 531
View_LineColor 532
ViewElement 467
Viewport.Clear (method) 884
Viewport.Close (method) 885
Viewport.Open (method) 886
Virtual Path for Scripts 15
Virtual Path Map 16
Visual Basic Compatibility 1220
VLine (statement) 1183
VPage (statement) 1183
VScroll (statement) 1184

W

Watch Expressions 45
Watch List
  selecting variables 46
Watch Variables
  deleting 46
Weekday (function) 845
What Is an Object 1228
While...Wend (statement) 1185
Wildcards 1051
WinActivate (statement) 1187

WinClose (statement) 1188
WinFind (function) 847
WinList (statement) 1189
WinMaximize (statement) 1190
WinMinimize (statement) 1192
WinMove (statement) 1193
WinRestore (statement) 1194
WinSize (statement) 1196
Word$ (function) 848
WordCount (function) 849
Working with Collections 29
Working with Model Properties 17
Working with Rational Rose RealTime Diagrams 17
Working with the Dialog Editor 48
Working with the Rose RealTime Script Editor 33
Working with Watch Variables 44
Workspace 128
WriteIni (statement) 1197

X

Xor (operator) 911

Y

Year (function) 850