



Wonderware  
Application Server  
Scripting Guide

11/17/15

All rights reserved. No part of this documentation shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Schneider Electric Software, LLC. No copyright or patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this documentation, the publisher and the author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

The information in this documentation is subject to change without notice and does not represent a commitment on the part of Schneider Electric Software, LLC. The software described in this documentation is furnished under a license agreement. This software may be used or copied only in accordance with such license agreement.

© 2015 Schneider Electric Software, LLC. All rights reserved.

Schneider Electric Software, LLC  
26561 Rancho Parkway South  
Lake Forest, CA 92630 U.S.A.  
(949) 727-3200

<http://software.schneider-electric.com>

For comments or suggestions about the product documentation, send an e-mail message to [ProductDocumentationComments@schneider-electric.com](mailto:ProductDocumentationComments@schneider-electric.com).

ArchestrA, Avantis, DYN SIM, EYESIM, Foxboro, Foxboro Evo, I/A Series, InBatch, InduSoft, IntelaTrac, InTouch, PIPEPHASE, PRO/II, PROVISION, ROMeo, Schneider Electric, SIM4ME, SimCentral, SimSci, Skelta, SmartGlance, Spiral Software, VISUAL FLARE, WindowMaker, WindowViewer, and Wonderware are trademarks of Schneider Electric SE, its subsidiaries, and affiliated companies. An extensive listing of Schneider Electric Software, LLC trademarks can be found at: <http://software.schneider-electric.com/legal/trademarks/>. All other brands may be trademarks of their respective owners.

License for Scintilla and SciTE

Copyright 1998-2002 by Neil Hodgson <[neilh@scintilla.org](mailto:neilh@scintilla.org)> All Rights Reserved

Neil Hodgson disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness, in no event shall neil hodgson be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

---

# Contents

	Welcome .....	9
	Documentation Conventions .....	9
	Technical Support .....	10
<b>Chapter 1</b>	<b>Common Scripting Environment .....</b>	<b>11</b>
	Script Editing Styles and Syntax .....	11
	Required Syntax for Expressions and Scripts .....	12
	Simple Scripts .....	12
	Script Execution Types .....	13
	Startup Scripts .....	13
	OnScan Scripts .....	13
	Execute Scripts .....	13
	OffScan Scripts .....	14
	Shutdown Scripts .....	15
	Deployment Scripts .....	15
	Dynamic Referencing Considerations .....	16
	Run-Time Client Script Behavior .....	18
	Opening a Client Application Window .....	18
	Closing a Client Application Window .....	18
	Minimizing a Client Application Window .....	19
	Maximizing or Restoring a Client Application Window .....	19
	Working with QuickScript Editor Features .....	19

---

Color Indicators for Script Elements .....	19
Autocomplete .....	20
Accepting Autocomplete Suggestions .....	24
Multi-level Undo and Redo .....	24
Visual Indication of Script Errors .....	24
Line Numbers .....	25
Log Functions .....	25
<b>Chapter 2 QuickScript .NET Functions.....</b>	<b>27</b>
Script Functions .....	28
Abs() .....	28
ActivateApp() .....	28
Filtering Events .....	29
ArcCos() .....	30
ArcSin() .....	31
ArcTan() .....	31
Cos() .....	32
CreateObject() .....	32
DateTimeGMT() .....	33
DText() .....	33
Exp() .....	34
GetCPQuality .....	34
GetCPTimeStamp .....	35
HideGraphic() .....	35
HideSelf() .....	36
Int() .....	36
IsBad() .....	37
IsGood() .....	37
IsInitializing() .....	38
IsUncertain() .....	39
IsUsable() .....	39
Log() .....	40
LogN() .....	41
Log10() .....	41
LogDataChangeEvent() .....	42
LogCustom() .....	42
LogError() .....	43
LogMessage() .....	44
LogTrace() .....	45
LogWarning() .....	46
Now() .....	47
Pi() .....	47

---

Round()	47
SendKeys()	48
SetAttributeVT()	50
SetBad()	51
SetGood()	51
SetInitializing()	52
SetUncertain()	52
Sgn()	53
ShowGraphic()	54
SignedAlarmAck()	65
SignedWrite()	70
Sin()	75
Sqrt()	75
StringASCII()	76
StringChar()	76
StringCompare()	77
StringCompareNoCase()	78
StringFromGMTTimeToLocal()	78
StringFromIntg()	80
StringFromReal()	80
StringFromTime()	81
StringFromTimeLocal()	82
StringInString()	83
StringLeft()	85
StringLen()	85
StringLower()	86
StringMid()	87
StringReplace()	88
StringRight()	89
StringSpace()	90
StringTest()	90
StringToIntg()	92
StringToReal()	92
StringTrim()	93
StringUpper()	94
Tan()	95
Text()	95
Trunc()	96
WriteStatus()	97
WWControl()	98
WWExecute()	98
WWPoke()	100
WWRequest()	101

WWStringFromTime() .....	102
QuickScript .NET Variables .....	103
Numbers and Strings .....	107
QuickScript .NET Control Structures .....	108
IF ... THEN ... ELSEIF ... ELSE ... ENDIF .....	108
IF ... THEN ... ELSEIF ... ELSE ... ENDIF and Attribute Quality .....	110
FOR ... TO ... STEP ... NEXT Loop .....	111
FOR EACH ... IN ... NEXT .....	113
TRY ... CATCH .....	113
WHILE Loop .....	115
QuickScript .NET Operators .....	115
Parentheses ( ) .....	117
Negation ( - ) .....	117
Complement ( ~ ) .....	117
Power ( ** ) .....	117
Multiplication ( * ), Division ( / ), Addition ( + ), Subtraction ( - ) .....	118
Modulo (MOD) .....	118
Shift Left (SHL), Shift Right (SHR) .....	118
Bitwise AND ( & ) .....	119
Exclusive OR ( ^ ) and Inclusive OR (   ) .....	119
Assignment ( = ) .....	119
Comparisons ( <, >, <=, >=, ==, <> ) .....	119
AND, OR, and NOT .....	120
Chapter 3 Sample QuickScript .NET Scripts .....	121
Sample Scripts .....	121
Accessing an Excel Spreadsheet Using an Imported Type Library .....	122
Accessing an Excel Spreadsheet Using CreateObject .....	123
Accessing an Office XP Excel Spreadsheet Using an Imported Type Library .....	123
Calling a Web Service to Get the Temperature for a Specified Zip Code .....	124
Calling a Web Service to Send an E-mail Message .....	124
Creating a Look-up Table and Doing a Look-up on It .....	125
Creating an XML Document and Saving it to Disk .....	125
Executing a SQL Parameterized INSERT Command .....	127
Filling a String Array and Using It .....	128
Filling a Two-Dimensional Integer Array and Using It .....	128
Formatting a Number Using a .NET Format 'Picture' .....	128

---

Formatting a Time Using a .NET Format 'Picture' .....	129
Getting the Directories Under the C Drive .....	129
Loading an XML Document from Disk and Doing Look-ups on It .....	129
Querying a SQL Server Database .....	130
Reading a Performance Counter .....	130
Reading a Text File from Disk .....	131
Sharing a SQL Connection or Any Other .NET Object .....	131
Using DDE to Access an Excel Spreadsheet .....	132
Using Microsoft Exchange to Send an E-mail Message .....	132
Using Screen-Scraping to Get the Temperature for a City .....	133
Using SMTP to Send an E-mail Message .....	133
Writing a Text File to Disk .....	134
Dynamically Binding an Indirect Variable to a Reference .....	134
Glossary .....	137
Index .....	143





---

# Welcome

This guide explains how to write Application Server scripts. This guide does not explain programming concepts; rather it is a reference for you after you learned the basics of scripting in Application Server.

You should understand standard programming techniques before writing Application Server scripts. If you do not know how to program in any language, contact Wonderware® or your distributor for information about training.

For more information on using Application Server, see the *Application Server User's Guide*.

You can view this document online or you can print it, in part or whole, in Adobe Reader.

## Documentation Conventions

This documentation uses the following conventions:

<b>Convention</b>	<b>Used for</b>
Initial Capitals	Paths and file names.
<b>Bold</b>	Menus, commands, dialog box names, and dialog box options.
Monospace	Code samples and display text.

## Technical Support

Wonderware Technical Support offers a variety of support options to answer any questions on Wonderware products and their implementation.

Before you contact Technical Support, refer to the relevant section(s) in this documentation for a possible solution to the problem. If you need to contact technical support for help, have the following information ready:

- The type and version of the operating system you are using.
- Details of how to recreate the problem.
- The exact wording of the error messages you saw.
- Any relevant output listing from the Log Viewer or any other diagnostic applications.
- Details of what you did to try to solve the problem(s) and your results.
- If known, the Wonderware Technical Support case number assigned to your problem, if this is an ongoing problem.

# Chapter 1

## Common Scripting Environment

This section describes common styles, syntax, commands, and behaviors of Application Server scripts.

### Script Editing Styles and Syntax

Application Server supports two types of scripts:

- Simple scripts can perform assignments, comparisons, simple math functions, and similar actions. Simple scripts are described in this section.
- Complex scripts can perform logical operations using conditional branching with IF-THEN-ELSE type control structures. For more information about complex control structures, see “QuickScript .NET Control Structures” on page 108.

Both single and multi-line comments are supported. Single-line comments start with a " ' " in the line but require no ending " ' " in the line. Multi-line comments start with a "{" and end with a "}" and can span multiple lines.

White space rules apply for space and indentation. Indent using spaces, or the TAB key. Individual statements are indicated by a semicolon marking the end of the statement.

## Required Syntax for Expressions and Scripts

The syntax in scripts is similar to the algebraic syntax of a calculator. Most statements are presented using the following form:

```
a = (b - c) / (2 + x) * xyz;
```

This statement places the value of the expression to the right of the equal sign (=) in the variable location named “a.”

- A single entity must appear to the left of the assignment operator =.
- The operands in an expression can be constants or variables.
- Statements must end with a semicolon (;).

Entities can be concatenated by using the plus (+) operator. For example, if a data change script such as the one below is created, each time the value of “Number” changes, the indirect entity “Setpoint” changes accordingly:

```
Number=1;  
Setpoint = "Setpoint" + Text(Number, "#");
```

Where the result is “Setpoint1.”

## Simple Scripts

Simple scripts implement logic such as assignments, math, and functions. An example of this type of scripting is:

```
React_temp = 150;  
ResultTag = (Sample1 + Sample2)/2;  
{this is a comment}
```

# Script Execution Types

This section describes the script execution types within Application Server.

## Startup Scripts

Startup scripts are called when an object containing the script is loaded into memory, such as during deployment, platform, or engine start.

Startup instantiates COM objects and .NET objects. Depending on load and other factors, assignments to object attributes from the Startup method may fail. Attributes that reside off-object are not available to the Startup method.

## OnScan Scripts

OnScan scripts are called the first time an AppEngine calls this object to execute after the object's scan state changes to OnScan. The OnScan method initiates local object attribute values or provides more flexibility in the creation of .NET or COM objects.

Attributes that are off-engine are not available to the OnScan method.

## Execute Scripts

Execute scripts are called each time the AppEngine performs a scan and the object is OnScan.

The Execute script method is the workhorse of the scripting execution types. Use the Execute method for your run-time scripting to ensure that all attributes and values are available to the script.

If the quality check-box is checked, the Execute method is similar to InTouch® scripts with the following conditional trigger types:

- **Periodic:** When going OnScan, a script with a periodic trigger executes immediately (at the next scheduled scan period of the AppEngine). It then executes periodically whenever the elapsed time evaluates as true.
- **Data Change:** Executes when a data value or quality changes between scans.

For the following trigger types, data changes between each scan are not evaluated, only the value at the beginning of each script is used for evaluation purposes. For example, if a Boolean attribute changes from True to False to True again during a scan cycle, this change is not evaluated as a data change as the value is True at the beginning of each scan cycle.

- **OnTrue:** Executes if the expression validates from a false on one scan to a true on the next scan.
- **OnFalse:** Executes if the expression validates from a true on one scan to a false on the next scan.

These scripts also have time-based considerations. A trigger period of 0 means that the script executes every scan.

Time-based scripts, **WhileTrue**, **WhileFalse**, and **Periodic** are evaluated and executed based on the elapsed time from a timestamp generated from the previous execution, not on an elapsed time counter. It is possible that a change in the system clock can change the interval between execution of these scripts.

- **WhileTrue:** Executes scan to scan as long as the expression validates as true at the beginning of the scan.
- **WhileFalse:** Executes scan to scan as long as the expression validates as false at the beginning of the scan.

For example, a periodic script is set to run every 60 minutes. The script executes at 11:13 AM. We expect it to execute 60 minutes later at 12:13 PM. However, a time synchronization event occurred and the node's time is adjusted from 11:33 AM to 11:30 AM.

The script still executes when the system time reaches 12:13 PM. But because of the time change, the actual (True) time period that elapsed between executions is 63 minutes.

## OffScan Scripts

OffScan scripts are called when the object is taken OffScan. This script type is primarily used to clean up the object and account for any needs to address as a result of the object no longer executing.

If an object is taken OffScan, either directly, or indirectly because its engine is taken OffScan, all in-progress asynchronous scripts for that object are requested to shut down by setting a Boolean shutdown attribute for the script to true. A well-written script checks this attribute before and after time-consuming operations. If the script takes more than 30 seconds to complete, a warning appears in the logger that the script is not responding to the shutdown command. However, the script is allowed to complete and is not terminated by force. This all takes place on the engine's main thread and could potentially hang the engine. During this time, the script might also time out and as a result exit before executing all its logic.

## Shutdown Scripts

Shutdown scripts are called when the object is about to be removed from memory, usually as a result of the AppEngine stopping. Shutdown scripts are primarily used to destroy COM objects and .NET objects and to free memory.

## Deployment Scripts

Deploying objects is both a critical and a load-intensive process for a Galaxy. Implementing scripting in the Startup and OnScan methods can adversely affect a Galaxy's deployment and redundancy performance.

While objects are being deployed, their Startup and, if deployed OnScan scripts are executed. These scripts must complete within the deployment time-out period for the deployment to be successful.

Placing large numbers of scripts, or scripts that require heavy processing power into the Startup or OnScan script methods can slow or cause a deployment or failover to fail. In addition to the load that is placed on the system at deployment time, the type of scripting done in the Startup and OnScan methods is also important because these scripts execute in a sequence.

During deployment and restart, the Startup and OnScan script methods do not execute objects based on execution order. Objects are started up and placed on scan based on their alphanumeric tag name within their hosting Area.

Follow the recommendation below for each type of script method to help determine what scripting practices to follow in each script method.

Do not place the following types of scripting in the Startup or OnScan methods:

- Database access
- File system access, .csv, .xml, .txt, and so on
- Off-object referencing
- Dynamic referencing

## Dynamic Referencing Considerations

Dynamic reference scripting is one of the biggest causes of deployment failures. It is one of the most common misuses of the Startup and OnScan methods.

Rather than placing dynamic referencing scripts in the Startup or OnScan methods, perform dynamic referencing in the Execute method. There are several advantages to using the Execute method with dynamic reference scripting:

- Deployment is faster.
- Deployment is more reliable.
- Deterministic execution order is guaranteed.
- Off-object and off-engine attributes are available.
- After a failover occurs, the startup of the redundant engine is more stable and can be faster.

### To create a simple dynamic reference script example

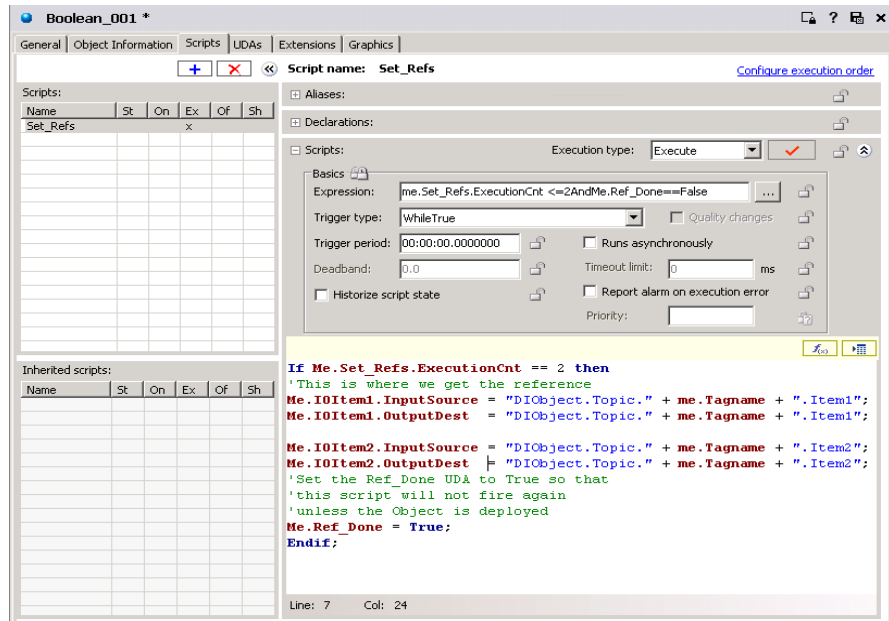
- 1 Create a Boolean attribute.

The screenshot shows the 'Reference\_Example' configuration window. The 'UDAs' tab is selected, displaying a table of UDAs. The table has columns for 'Name', 'IO\_Item1', 'IO\_Item2', and 'Ref\_Done'. The 'IO\_Item1' row is highlighted. To the right of the table, the 'UDA name' is set to 'IO\_Item1', the 'Data type' is 'Boolean', and the 'Category' is 'User writeable'. There are also checkboxes for 'This is an array' and 'True / False'.

The attribute shows if the referencing script is complete. In this example you create Ref\_Done. IO\_Item1 and IO\_Item2 are the I/O points referenced in this example.



- 2 Create the script. The script in this example is called `Set_Refs`. The script has a trigger type of `WhileTrue` with a 0 trigger period.



The script is shown below:

```

If Me.Set_Refs.ExecutionCnt == 2 then
  Me.IOItem1.InputSource = "DIObject.Topic." + me.Tagname +
  ".Item1";
  Me.IOItem1.OutputDest = "DIObject.Topic." + me.Tagname +
  ".Item1";
  Me.IOItem2.InputSource = "DIObject.Topic." + me.Tagname +
  ".Item2";
  Me.IOItem2.OutputDest = "DIObject.Topic." + me.Tagname +
  ".Item2";
  Me.Ref_Done = True;
Endif;
  
```

This script allows the system to stabilize after going on scan before setting the references. The script executes on the first two scans of the object when the Boolean attribute `Ref_Done` is false.

As the script is executed, a check is made against the execution count. If the count equals 2, the script performs the referencing operations. After the reference attributes are set on the attributes, the `Ref_Done` attribute is set to `True`. At this point the expression for the script is no longer true.

The three attributes set in this script are checkpointed, eliminating the need to run this script except on deployment. The next time the object is started, placed on scan, or failed over, there is no need to recreate the references to the items.

## Run-Time Client Script Behavior

In Advanced Communication Management, script references to InTouch tags and object attributes are suspended from receiving data changes when the application window containing embedded ArcestrA® objects is minimized in InTouch WindowViewer. Suspending data updates to hidden objects reduces the amount of network traffic and improves the overall performance of a client application.

While Showing scripts of embedded symbols do not execute during the period when the window containing the symbols is minimized. Script execution resumes after restoring or maximizing a window that had been previously closed or minimized.

## Opening a Client Application Window

In Advanced Communication Management, when a client application window containing embedded ArcestrA objects opens in WindowViewer, the following script events occur:

- Register all ArcestrA and InTouch references used in embedded symbol scripts, if not registered already.
- Advise all ArcestrA and InTouch references in embedded symbol scripts within the window, if not advised already.
- Execute the OnShow script on all embedded symbol scripts within the window.
- Execute named scripts if their trigger conditions are met.

## Closing a Client Application Window

In Advanced Communication Management, when a client application window containing embedded ArcestrA objects is closed, the following script events occur:

- Execute OnHide scripts of all embedded symbols within the window.
- Stop running client scripts.
- Unadvise all ArcestrA and InTouch references in the Window if there are no other open windows using the references.
- Unregister all ArcestrA and InTouch references in the Window if there are no other open windows using the references.

## Minimizing a Client Application Window

In Advanced Communication Management, when an open window containing embedded ArcestrA objects is minimized in WindowViewer, the following script events occur:

- Stop running client scripts associated with ArcestrA objects embedded in the window.
- Unadvise all ArcestrA and InTouch references in the Window if there are no other open windows using the references.
- OnHide scripts of embedded symbols do not execute when a window is minimized.

## Maximizing or Restoring a Client Application Window

In Advanced Communication Management, after maximizing or restoring a window from WindowViewer that had been previously minimized or closed, the following script events occur:

- Advise all ArcestrA and InTouch script references in the window, if not advised already.
- Execute named scripts if their trigger conditions are met.

## Working with QuickScript Editor Features

The QuickScript editor provides a number of features to enhance scripting speed and accuracy.

### Color Indicators for Script Elements

The QuickScript .NET editor uses different text colors to identify different script elements. The following table shows the text colors associated with script elements.

Element	Color
Keywords	Blue Syntax highlighted while typing.
Comments (both single line and multi-line)	Green Syntax highlighted while typing.
Strings	Purple Syntax highlighted while typing.

<b>Element</b>	<b>Color</b>
Function names, numeric constants, operators, semicolons, dim variables, alias variables, and so on	Black See descriptions for Attribute names and Reserved words.
Attributes, InTouch Tags, Reference Strings	Maroon, bold face
Reserved words	Red, non-bold face
.NET type names	Teal, non-bold face



## Autocomplete



















QuickScript autocomplete incorporates several features for use while authoring object and client scripts:














- Provides an autocomplete Attribute reference when you type a generic object name, such as "me." Run-time attributes appear in an autocomplete list box. Typing "InTouch:" displays an autocomplete list of tagnames from the most recently selected ViewApp template.
- Provides method parameter help in an autocomplete list box including context-specific suggestions covering definitions, keywords, script elements, and programmatic constructs such as try ... catch or while ... endwhile.
- Automatic word completion of Attribute references, methods, programmatic constructs, and other script elements.


















These features serve as convenient documentation of method parameters and scripting syntax as well as an enhanced input method.





Autocomplete displays a context-sensitive list of options for script elements, keywords, object and attribute names, and programmatic constructs. Press CONTROL+space to display all available autocomplete options and variables for the selected location in the script. You can identify the context from the icons displayed with the list items.

<b>Icon</b>	<b>Represents</b>
	MxBoolean attribute
	MxInteger attribute

Icon	Represents
	MxFloat attribute
	MxDouble attribute
	MxString attribute
	MxTime attribute
	MxElapsedTime attribute
	MxReference attribute
	MxStatus attribute
	MxDatatypeEnum attribute
	MxSecurityClassification attribute
	MxDataQuality attribute
	MxQualifiedEnum attribute
	MxQualifiedStruct attribute
	MxInternationalizedString attribute
	.Net Method
	.Net Property
	.Net Field or Variable
	.Net Namespace
	.Net Struct

Icon	Represents
	.Net Class
	.Net Interface
	.Net Enumeration
	.Net Enum Value
	QuickScript Keyword
	<p>Contained object name, or any partial attribute name such as a attribute, field attribute, or primitive that has a dot in the name, or any attribute of Mx type MxNone, or if there are several type choices among objects and attributes.</p> <p>If the attribute cannot be exactly or unambiguously returned, this icon will appear.</p> <p>Partial name example: For <b>me.alarm.a1</b>, typing "me.alar" will show the blue ball icon for alarm.</p> <p>MxNone example: input/output extension attribute WriteValue.</p>
	Rectangle
	Rounded rectangle
	Line
	Horizontal or vertical line
	Text
	Ellipse
	Curve

Icon	Represents
	Closed curve
	Button
	Polygon
	Polyline
	Connect
	Image
	Group or embedded symbol
	Alarm control
	Edit box
	Arc
	Pie
	Chord
	Circle
	Status
	Radio buttons
	Checkbox
	Edit box

Icon	Represents
	Combo box
	Calendar
	Date picker
	List box

## Accepting Autocomplete Suggestions

Insert an item at the editor caret from the autocomplete list box—without an end line or tab appended—by doing one of the following:

- Double-click the item
- Highlight (select) the item and press the **Enter** key or the **Tab** key.

Type a space, period, comma, open or closed parenthesis, or other punctuation used in the QuickScript .NET programming language (: ; [ ] = < > - + / \*), and the item highlighted in the autocomplete list box will be inserted at the editor caret with the additional character appended.

## Multi-level Undo and Redo

You can selectively undo a history of changes to your script. The number of changes that can be undone is limited only by the amount of available memory.

An undone change can be redone. Redo mirrors undo changes.

A single undo typically is comprised of sequences of typing or deleting, which can be interrupted by interaction with an autocomplete list or by moving the cursor with the mouse, or by clicking elsewhere in the script.

All pending undo and redo actions will be lost if you close the object editor, switch to another script within the object editor, or switch among Startup, OnScan, Execute, OffScan, and Shutdown scripts.

## Visual Indication of Script Errors

Verification errors in script text are marked with a red "squiggly" underline. The underline appears after approximately 2.5 seconds of keyboard inactivity.



Hovering over the error with the mouse cursor will display the error message as a tooltip. The tooltip error message is identical to that shown when clicking the script verification button.

---

**Note:** In addition to error tooltips, the script editor will also display the variable name and type in a tooltip when hovering over a variable name in the script.

---

In some cases, more than one error will be underlined. This is not always possible because some errors prevent the compiler from continuing past the error.

## Line Numbers

The script editor displays line numbers in the left margin.

- Line numbers of up to four digits will display when the script editor is not zoomed.
- The line number may appear clipped for scripts longer than 9999 lines or when the script editor is zoomed.
- Use the right-click context menu **Go To** function to go to a specific line in the script.

## Log Functions

QuickScript .NET functions include several log functions to capture and display information in the logger under different log flags.

- [LogCustom\(\)](#)
- [LogError\(\)](#)
- [LogMessage\(\)](#)
- [LogTrace\(\)](#)
- [LogWarning\(\)](#)

---

**Important:** To use the LogCustom function, you must enable Log Custom in the System Management Console (SMC) Log Flag Editor. To use the LogTrace function, you must enable Log Trace in the SMC Log Flag Editor.

---



## Chapter 2

# QuickScript .NET Functions

This section describes the script functions included in the Application Server development environment.

Functions are listed alphabetically with:

- A description
- The function category, as shown in the Script Function Browser
- The proper syntax with descriptions of parameters
- Examples

An additional category of script functions shown in the Script Function Browser are the Types functions, which are not described in this documentation. The Types functions include .NET functions provided by the Microsoft .NET Framework and any .NET functions developed with Microsoft Visual Studio .NET.

For descriptions of each function provided by the Microsoft .NET Framework, see the Microsoft Developers Network website:  
<http://msdn.microsoft.com/>

For information about other functions in this category, see third-party documentation.

Keep in mind the following limitations when you use the script functions:

- Be aware of the .NET datatypes.
- Starting a GUI application from within a server script is not supported.
- Although QuickScript supports import libraries built with .NET CLR version 2.0.50727, it does not support any of the new language features introduced with .NET 2.0, such as generics.

## Script Functions

Script functions are described in this section.

### Abs()

Returns the absolute value (unsigned equivalent) of a specified number.

#### Category

Math

#### Syntax

```
Result = Abs ( Number );
```

#### Parameter

*Number*

Any number or numeric attribute.

#### Examples

```
Abs(14); ' returns 14
```

```
Abs(-7.5); ' returns 7.5
```

### ActivateApp()

Restores, minimizes, maximizes, or closes another currently running Windows application.

---

**Note:** Microsoft Vista operating system security prevents services from interacting with desktop applications. Object scripts that include the ActivateApp() function do not work when running under Vista. A warning message is written to the logger. But, the ActivateApp() function does work successfully with *client scripts* on computers running Vista.

---

**Category**

Miscellaneous

**Syntax**

```
ActivateApp( TaskName );
```

**Parameter**

*TaskName*

The task this function activates.

**Remarks**

*TaskName* is the exact text string, including spaces, that appears on the Task Bar or in Windows Task Manager. You can see the task name by opening Task Manager.

**Example**

```
ActivateApp("Calculator");
```

## Filtering Events

To get only specific events, filters can be introduced before getting events from the event service. The filtering should be done before the `StartRequestingEvent()` method is called.

The following datatypes are supported when filtering the events.

- Integer
- Float
- String
- Bool
- DateTime
- Double
- Short
- Array

The following table shows the comparison types that are supported for filtering events.

<b>Comparison Keyword</b>	<b>Description</b>
eq	Means EqualTo. Returns all the events matching the filtered criteria.
beginswith	Means StartsWith. Returns all the events matching the filtered criteria. Applies only to string data type filtering
lt	Means Lesser Than. Applies to all supported data types excluding string. It does not support arrays.
le	Means Lesser or Equal. Applies to all supported data types excluding string. It does not support arrays.
gt	Means Greater Than. Applies to all supported data types excluding string. It does not support arrays.
ge	Means Greater or Equal. Applies to all supported data types excluding string. It does not support arrays.
between	Checks will be made only to paired supplied values. Returns all the events matching the filtered criteria. It supports numeric and date datatypes.
neg, nbegins, nlt, nle, ngt, nge, nbetween	A keyword 'n' before the comparison keyword Means NOT of.

## ArcCos()

Returns an angle between 0 and 180 degrees whose cosine is equal to the number specified.

### Category

Math

### Syntax

```
Result = ArcCos( Number );
```

**Parameter***Number*

Any number or numeric attribute with a value between -1 and 1 (inclusive).

**Examples**

```
ArcCos(1); ' returns 0
```

```
ArcCos(-1); ' returns 180
```

**See Also**

Cos(), Sin(), Tan(), ArcSin(), ArcTan()

## ArcSin()

Returns an angle between -90 and 90 degrees whose sine is equal to the number specified.

**Category**

Math

**Syntax**

```
Result = ArcSin( Number );
```

**Parameter***Number*

Any number or numeric attribute with a value between -1 and 1 (inclusive).

**Examples**

```
ArcSin(1); ' returns 90
```

```
ArcSin(-1); ' returns -90
```

**See Also**

Cos(), Sin(), Tan(), ArcCos(), ArcTan()

## ArcTan()

Returns an angle between -90 and 90 degrees whose tangent is equal to the number specified.

**Category**

Math

**Syntax**

```
Result = ArcTan( Number );
```

**Parameter***Number*

Any number or numeric attribute.

**Examples**

```
ArcTan(1); ' returns 45
```

```
ArcTan(0); ' returns 0
```

**See Also**

Cos(), Sin(), Tan(), ArcCos(), ArcSin()

## Cos()

Returns the cosine of an angle in degrees.

**Category**

Math

**Syntax**

```
Result = Cos( Number );
```

**Parameter***Number*

Any number or numeric attribute.

**Examples**

```
Cos(90); ' returns 0
```

```
Cos(0); ' returns 1
```

This example shows how to use the function in a math equation:

```
Wave = 50 * Cos(6 * Now().Second);
```

**See Also**

Sin(), Tan(), ArcCos(), ArcSin(), ArcTan()

## CreateObject()

Creates an ActiveX (COM) object.

**Category**

System

**Syntax**

```
ObjectResult = CreateObject( ProgID );
```



**Parameter***ProgID*

The program ID (as a string) of the object to be created.

**Example**

```
CreateObject("ADODB.Connection");
```

## DateTimeGMT()

Returns a number representing the number of days and fractions of days since January 1, 1970, in Coordinated Universal Time (UTC), regardless of the local time zone.

**Category**

Miscellaneous

**Syntax**

```
Result=DateTimeGMT();
```

**Parameters**

None

**Example**

```
MessageTag = StringFromTime(DateTimeGMT() * 86400.0, 3);
```

## DText()

Returns one of two possible strings, depending on the value of the *Discrete* parameter.

**Category**

String

**Syntax**

```
StringResult = DText( Discrete, OnMsg, OffMsg );
```

**Parameters***Discrete*

A Boolean value or Boolean attribute.

*OnMsg*

The message that is shown when the value of *Discrete* equals true.

*OffMsg*

The message shown when *Discrete* equals false.

**Example**

```
StringResult = DText(me.temp > 150, "Too hot", "Just right");
```

## Exp()

Returns the result of the exponent  $e$  raised to a power.

**Category**

Math

**Syntax**

```
Result = Exp( Number );
```

**Parameter**

*Number*

Any number or numeric attribute.

**Example**

```
Exp(1); ' returns 2.718...
```

## GetCPQuality

Returns the Quality value of a custom property. This function is available within any ArcestrA graphic client script.

**Syntax**

```
Int GetCPQuality(String name)
```

Where *String name* is the name of the custom property whose quality is to be retrieved.

This script function takes the name of a custom property on the symbol. This argument is of type string and it can be a reference or a constant.

If the custom property is type constant, GOOD is the quality always returned.

---

**Note:** For use with custom properties only. It does not apply to InTouch tags.

---

**Return Value**

The GetCPQuality() script function returns a value 0-255 of type Integer, as per the OPC quality standard. 192 is GOOD.

**Example**

```
cp2 = GetCPQuality("cp1");
```

Where cp1 and cp2 are custom properties and the data type of cp2 is Integer.

## GetCPTimeStamp

Returns the time stamp of a custom property. This function is available within any ArcestrA graphic client script.

### Syntax

```
DateTime GetCPTimeStamp(String name)
```

Where *String name* is the name of the custom property whose time stamp is to be retrieved.

This script function takes the name of a custom property on the symbol. This argument is of type string and it can be a reference or a constant.

---

**Note:** For use with custom properties only. It does not apply to InTouch tags.

---

### Return Value

The GetCPTimeStamp() script function returns the time stamp of the custom property's current value of type DateTime. If the custom property value is a constant, then the return value is the time the value was created.

### Example

```
cp2 = GetCPTimeStamp("cp1");
```

Where cp1 and cp2 are custom properties and the data type of cp2 is DateTime.

## HideGraphic()

Closes an open graphic pop-up window shown in the ShowGraphic() script with the given identity name.

The HideGraphic() function has been extended to close InTouch Windows identified with a given identity name. This function is available within any ArcestrA graphic client script.

### Category

Graphic Client

### Syntax

```
HideGraphic(string identity);
```

### Parameter

*Identity*

The unique name of the instance that shows the graphic.

**Examples**

```
HideGraphic("i1");
```

Where "i1" is string Identity.

```
HideGraphic("InTouch:Window1");
```

Where "InTouch:Window1" is the string identity.

**See Also**

ShowGraphic(), HideSelf()

## HideSelf()

Closes the displayed graphic for which this client script is configured. This script function is available within any ArcestrA graphic client script.

**Category**

Graphic Client

**Syntax**

```
HideSelf();
```

**Remarks**

You must call the script function within the symbol to hide the popup.

**Example**

```
HideSelf();
```

**See Also**

ShowGraphic(), HideGraphic()

## Int()

Returns the next integer less than or equal to a specified number.

**Category**

Math

**Syntax**

```
IntegerResult = Int( Number );
```

**Parameter**

*Number*

Any number or numeric attribute.

**Remarks**

When handling negative real (float) numbers, this function returns the integer farthest from zero.

**Examples**

```
Int(4.7); ' returns 4
```

```
Int(-4.7); ' returns -5
```

## IsBad()

Returns a Boolean value indicating if the quality of the specified attribute is Bad.

**Category**

Miscellaneous

**Syntax**

```
BooleanResult = IsBad( Attribute1, Attribute2, ... );
```

**Parameter(s)**

*Attribute1*, *Attribute2*, ...*AttributeN*

Names of one or more attributes for which you want to determine Bad quality. You can include a variable-length list of attributes.

**Return Value**

If any of the specified attributes has Bad quality, then true is returned. Otherwise, false is returned.

**Examples**

```
IsBad(TIC101.PV);
```

```
IsBad(TIC101.PV, PIC102.PV);
```

**See Also**

IsGood(), Initializing(), IsUncertain(), IsUsable()

## IsGood()

Returns a Boolean value indicating if the quality of the specified attribute is Good.

**Category**

Miscellaneous

**Syntax**

```
BooleanResult = IsGood( Attribute1, Attribute2, ... );
```

**Parameter(s)**

*Attribute1, Attribute2, and so on*

Name of the attribute(s) for which you want to determine Good quality. You can include a variable-length list of attributes.

**Return Value**

If all of the specified attributes have Good quality, then true is returned. Otherwise, false is returned.

**Examples**

```
IsGood(TIC101.PV);
```

```
IsGood(TIC101.PV, PIC102.PV);
```

**See Also**

IsBad(), IsInitializing(), IsUncertain(), IsUsable()

## IsInitializing()

Returns a Boolean value indicating if the quality of the specified attribute is Initializing.

**Category**

Miscellaneous

**Syntax**

```
BooleanResult = IsInitializing( Attribute1, Attribute2, ... );
```

**Parameter(s)**

*Attribute1, Attribute2, and so on*

Name of the attribute(s) for which to determine Initializing quality. You can include a variable-length list of attributes.

**Return Value**

If any of the specified attributes has Initializing quality, then true is returned. Otherwise, false is returned.

**Examples**

```
IsInitializing(TIC101.PV);
```

```
IsInitializing(TIC101.PV, PIC102.PV);
```

**See Also**

IsBad(), IsGood(), IsUncertain(), IsUsable()

## IsUncertain()

Returns a Boolean value indicating if the quality of the specified attribute is Uncertain.

### Category

Miscellaneous

### Syntax

```
BooleanResult = IsUncertain( Attribute1, Attribute2, ... );
```

### Parameter(s)

*Attribute1, Attribute2, and so on*

Name of the attribute(s) to determine Uncertain quality. You can include a variable-length list of attributes.

### Return Value

If all of the specified attributes have Uncertain quality, then true is returned. Otherwise, false is returned.

### Examples

```
IsUncertain(TIC101.PV);
```

```
IsUncertain(TIC101.PV, PIC102.PV);
```

### See Also

IsBad(), IsGood(), IsInitializing(), IsUsable()

## IsUsable()

Returns a Boolean value indicating if the specified attribute is usable for calculations.

### Category

Miscellaneous

### Syntax

```
BooleanResult = IsUsable( Attribute1, Attribute2, ... );
```

### Parameter(s)

*Attribute1, Attribute2, ...AttributeN*

Name of one or more attributes for which you want to determine unusable quality. You can include a variable-length list of attributes.

**Return Value**

If all of the specified attributes have either Good or Uncertain quality, then true is returned. Otherwise, false is returned.

**Remarks**

To qualify as usable, the attribute must have Good or Uncertain quality. In addition, each float or double attribute cannot be a NaN (not a number).

**Examples**

```
IsUsable(TIC101.PV);  
IsUsable(TIC101.PV, PIC102.PV);
```

**See Also**

IsBad(), IsGood(), IsInitializing(), IsUncertain()

## Log()

Returns the natural log (base e) of a number.

**Category**

Math

**Syntax**

```
RealResult = Log( Number );
```

**Parameter**

*Number*

Any number or numeric attribute.

**Remarks**

Natural log of 0 is undefined.

**Examples**

```
Log(100); ' returns 4.605...  
Log(1); ' returns 0
```

**See Also**

LogN(), Log10()



## LogN()

Returns the values of the logarithm of x to base n.

### Category

Math

### Syntax

```
Result = LogN( Number, Base );
```

### Parameters

*Number*

Any number or numeric attribute.

*Base*

Integer to set log base. You could also specify an integer attribute.

### Remarks

Base 1 is undefined.

### Examples

```
LogN(8, 3); ' returns 1.89279
```

```
LogN(3, 7); ' returns 0.564
```

### See Also

Log(), Log10()

## Log10()

Returns the base 10 log of a number.

### Category

Math

### Syntax

```
Result = Log10( Number );
```

### Parameter

*Number*

Any number or numeric attribute.

### Example

```
Log10(100); ' returns 2
```

### See Also

Log(), LogN()

## LogDataChangeEvent()

Logs an application change event to the Galaxy Historian.

---

**Note:** The LogDataChangeEvent() function works only in object scripts, not in symbol scripts.

---

### Category

Miscellaneous

### Syntax

```
LogDataChangeEvent (AttributeName, Description, OldValue,  
NewValue, TimeStamp);
```

### Parameters

*AttributeName*

Attribute name as a tag name.

*Description*

Description of the object.

*OldValue*

Old value of the attribute.

*NewValue*

New value of the attribute.

*TimeStamp*

The time stamp associated with the logged event. The timestamp can be UTC or local time. The TimeStamp parameter is optional. The timestamp of the logged event defaults to Now() if a TimeStamp parameter is not included.

### Remarks

A symbol script still compiles if the LogDataChangeEvent() function is included. However, a warning message is written to the log at run time that the function is inoperable.

### Example

This example logs an event when a pump starts or stops with a timestamp of the current time when the event occurred.

```
LogDataChangeEvent (TC104.pumpstate, "Pump04", OldState,  
NewState);
```

## LogCustom()

Writes a user-defined custom flag message in the Log Viewer.

**Category**

Miscellaneous

**Syntax**

```
LogCustom( CustomFlag, msg );
```

**Parameter***CustomFlag*

Creates a new log flag based on the first parameter string. The first call creates the custom flag.

*msg*

The message to write to the Log Viewer. Actual string or a string attribute.

**Remarks**

The log flag is disabled by default.

The message is always logged under the component "ObjectName.ScriptName". For example, "WinPlatform\_001.script1:msg", which identifies what object and what script within the object logged the error.

LogCustom() is similar to LogMessage(), but displays the message in the custom log flag when Log Custom is enabled.

The parameter help tooltip and Function Browser sample parameter list will show "LogCustom( CustomFlag, msg )" rather than "LogCustom( CustomFlag, Message )". "Message" is a reserved keyword.

**Example**

```
LogCustom(EditBox1.text, "User-defined message.");
```

This statement writes to the Log Viewer as follows:

```
10/24/2005 12:49:14 PM ScriptRuntime
```

```
<ObjectName.ScriptName>: <LogFlag EditBox1> User-defined  
message.
```

## LogError()

Writes a user-defined error message in the Log Viewer with a red error log flag.

**Category**

Miscellaneous

**Syntax**

```
LogError( msg );
```

**Parameter***msg*

The message to write to the Log Viewer. Actual string or a string attribute.

**Remarks**

The log flag is enabled by default.

The message is always logged under the component "ObjectName.ScriptName". For example, "WinPlatform\_001.script1:msg", which identifies what object and what script within the object logged the error.

LogError() is similar to LogMessage(), but displays the message in red.

The parameter help tooltip and Function Browser sample parameter list will show "LogError( msg )" rather than "LogError( Message )". "Message" is a reserved keyword.

**Example**

```
LogError("User-defined error message.");
```

This statement writes to the Log Viewer as follows:

```
10/24/2005 12:49:14 PM ScriptRuntime
```

```
<ObjectName.ScriptName>: User-defined error message.
```

## LogMessage()

Writes a user-defined message to the Log Viewer.

**Category**

Miscellaneous

**Syntax**

```
LogMessage( msg );
```

**Parameter***msg*

The message to write to the Log Viewer. Actual string or a string attribute.

**Remarks**

This is a very powerful function for troubleshooting scripting. By strategically placing LogMessage() functions in your scripts, you can determine the order of script execution, performance of scripts, and identify the value of attributes both before they are changed and after they are affected by the script.

Each message posted to the Log Viewer is stamped with the exact date and time. The message always begins with the component "TagName.ScriptName" so you can tell what object and what script within the object posted the message to the log.

### Examples

```
LogMessage("Report Script is Running");
```

The above statement writes the following to the Log Viewer:

```
10/24/2005 12:49:14 PM ScriptRuntime
  <TagName.ScriptName>:Report Script is Running.

MyTag=MyTag + 10;

LogMessage("The Value of MyTag is " + Text(MyTag, "#"));
```

## LogTrace()

Writes a user-defined trace message in the Log Viewer.

### Category

Miscellaneous

### Syntax

```
LogTrace( msg );
```

### Parameter

*msg*

The message to write to the Log Viewer. Actual string or a string attribute.

### Remarks

The log flag is disabled by default.

The message is always logged under the component "ObjectName.ScriptName". For example, "WinPlatform\_001.script1:msg", which identifies what object and what script within the object logged the error.

LogTrace() is similar to LogMessage(), but displays the message as Trace when Log Trace is enabled.

The parameter help tooltip and Function Browser sample parameter list will show "LogTrace( msg )" rather than "LogTrace( Message )". "Message" is a reserved keyword.

**Example**

```
LogTrace("User-defined trace message.");
```

This statement writes to the Log Viewer as follows:

```
10/24/2005 12:49:14 PM ScriptRuntime  
<ObjectName.ScriptName>: User-defined trace message.
```

## LogWarning()

Writes a user-defined error message in the Log Viewer with a yellow warning log flag.

**Category**

Miscellaneous

**Syntax**

```
LogWarning( msg );
```

**Parameter**

*msg*

The message to write to the Log Viewer. Actual string or a string attribute.

**Remarks**

The log flag is disabled by default.

The message is always logged under the component "ObjectName.ScriptName". For example, "WinPlatform\_001.script1:msg", which identifies what object and what script within the object logged the error.

LogWarning() is similar to LogMessage(), but displays the message as a yellow warning message.

The parameter help tooltip and Function Browser sample parameter list will show "LogWarning( msg )" rather than "LogWarning( Message )". "Message" is a reserved keyword.

**Example**

```
LogWarning("User-defined warning message.")
```

This statement writes to the Log Viewer as follows:

```
10/24/2005 12:49:14 PM ScriptRuntime  
<ObjectName.ScriptName>: User-defined warning message.
```

## Now()

Returns the current time.

### Category

System

### Syntax

```
TimeValue = Now();
```

### Remarks

The return value can be formatted using .NET functions.

## Pi()

Returns the value of Pi.

### Category

Math

### Syntax

```
RealResult = Pi();
```

### Example

```
Pi(); ' returns 3.1415926
```

## Round()

Rounds a real number to a specified precision and returns the result.

### Category

Math

### Syntax

```
RealResult = Round( Number, Precision );
```

### Parameters

*Number*

Any number or numeric attribute.

*Precision*

Sets the precision to which the number is rounded. This value can be any number or a numeric attribute.

**Examples**

```
Round(4.3, 1); ' returns 4
Round(4.3, .01); ' returns 4.30
Round(4.5, 1); ' returns 5
Round(-4.5, 1); ' returns -4
Round(106, 5); ' returns 105
Round(43.7, .5); ' returns 43.5
```

**See Also**

Trunc()

## SendKeys()

Sends keystrokes to an application. To the receiving application, the keys appear to be entered from the keyboard. You can use SendKeys() within a script to enter data or send commands to an application. Most keyboard keys can be used in a SendKeys () statement. Each key is represented by one or more characters, such as A for the letter A or {ENTER} for the Enter key.

---

**Caution:** Microsoft Vista or later operating system security prevents services from interacting with desktop applications. Object scripts that include the SendKeys() function do not work when running under these operating systems. A warning message is written to the logger. But, the SendKeys() function does work successfully with client scripts on computers running these systems.

---

**Category**

Miscellaneous

**Syntax**

```
SendKeys ( KeySequence );
```

**Parameter**

*KeySequence*

Any key sequence or a string attribute.

**Remarks**

To specify more than one key, concatenate the codes for each character. For example, to specify the dollar sign (\$) key followed by a (b), enter \$b.



The following lists the valid send key codes for unique keyboard keys:

<b>Key</b>	<b>Code</b>
BACKSPACE	{BACKSPACE} or {BS}
BREAK	{BREAK}
CAPSLOCK	{CAPSLOCK}
DELETE	{DELETE} or {DEL}
DOWN	{DOWN}
END	{END}
ENTER	{ENTER} or tilde (~)
ESCAPE	{ESCAPE} or {ESC}
F1...F12	{F1}...{F12}
HOME	{HOME}
INSERT	{INSERT}
LEFT	{LEFT}
NUMLOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
PRTSC	{PRTSC}
RIGHT	{RIGHT}
TAB	{TAB}
UP	{UP}
HOME	{HOME}

Special keys (SHIFT, CTRL, and ALT) have their own key codes:

<b>Key</b>	<b>Code</b>
SHIFT	+ (plus)
CTRL	^ (caret)
ALT	% (percent)

Enhancements to the Microsoft Hardware Abstraction Layer in Windows prevents the SendKeys() function from operating on some computers.

### Examples

To use two special keys together, use a second set of parentheses. The following statement holds down the CTRL key while pressing the ALT key, followed by p:

```
SendKeys ("^(%p))");
```

Commands can be preceded by the `ActivateApp()` command to direct the keystrokes to the proper application.

The following statement gives the computer focus to Calculator and sends the key combination 1234:

```
ActivateApp("Calculator");
```

```
SendKeys ("^(1234)");
```

## SetAttributeVT()

Sets the value and timestamp of an object attribute.

### Category

Miscellaneous

### Syntax

```
SetAttributeVT( Attribute, Value, TimeStamp);
```

### Parameter

#### *Attribute*

Name of the object attribute whose value and timestamp are modified. The specified attribute must belong to the object to which the script is attached.

#### *Value*

Value of the attribute, which can be a reference. The quality is always set to Good.

#### *TimeStamp*

Timestamp that can be a reference, a variable, or a string interpreted as the computer's local time or UTC. The timestamp is converted internally to UTC format before the attribute's value is sent to the run-time component.

### Remarks

Timestamp can be set only for object attributes that support a timestamp. At compile time, the script cannot detect whether the attribute specified with the `SetAttributeVT()` function supports a timestamp or not. No warning is issued if the attribute does not support a timestamp.

**Example**

This example sets an integer value and timestamp for an attribute that indicates pump RPM.

```
SetAttributeVT(me.PV, TC104.PumpRPM, LCLTIME);
```

## SetBad()

Sets the quality of an attribute to Bad.

**Category**

Miscellaneous

**Syntax**

```
SetBad( Attribute );
```

**Parameter**

*Attribute*

The attribute for which you want to set the quality to Bad.

**Remarks**

The specified attribute must be within the object to which the script is attached.

**Example**

```
SetBad(me.PV);
```

**See Also**

SetGood(), SetInitializing(), SetUncertain()

## SetGood()

Sets the quality of an attribute to Good.

**Category**

Miscellaneous

**Syntax**

```
SetGood( Attribute );
```

**Parameter**

*Attribute*

The attribute for which you want to set the quality to Good.

**Remarks**

The specified attribute must be within the object to which the script is attached.

**Example**

```
SetGood(me.PV);
```

**See Also**

SetBad(), SetInitializing(), SetUncertain()

## SetInitializing()

Sets the quality of an attribute to Initializing.

**Category**

Miscellaneous

**Syntax**

```
SetInitializing( Attribute );
```

**Parameter**

*Attribute*

The attribute for which you want to set the quality to Initializing.

**Remarks**

The specified attribute must be within the object to which the script is attached.

**Example**

```
SetInitializing(me.PV);
```

**See Also**

SetBad(), SetGood(), SetUncertain()

## SetUncertain()

Sets the quality of an attribute to Uncertain.

**Category**

Miscellaneous

**Syntax**

```
SetUncertain( Attribute );
```

**Parameter***Attribute*

The attribute for which you want to set the quality to Uncertain.

**Remarks**

The specified attribute must be within the object to which the script is attached.

**Example**

```
SetUncertain(me.PV);
```

**See Also**

SetBad(), SetGood(), SetInitializing()

## Sgn()

Determines the sign of a value (whether it is positive, zero, or negative) and returns the result.

**Category**

Math

**Syntax**

```
IntegerResult = Sgn( Number );
```

**Parameter***Number*

Any number or numeric attribute.

**Return Value**

If the input number is positive, the result is 1. Negative numbers return a -1, and 0 returns a 0.

**Examples**

```
Sgn(425); ' returns 1;
```

```
Sgn(0); ' returns 0;
```

```
Sgn(-37.3); ' returns -1;
```

## ShowGraphic()

Shows a graphic within a pop-up window. The ShowGraphic() function has been extended to call InTouch Windows. This function is available within any ArcestrA graphic client script.

### Category

Graphic Client

### Syntax

#### Show a graphic within a pop-up window

```
Dim graphicInfo as aaGraphic.GraphicInfo;  
graphicInfo.Identity = "<Identity>";  
graphicInfo.GraphicName = "<SymbolName>";  
ShowGraphic( graphicInfo );
```

#### Call an InTouch window

```
Dim graphicInfo as aaGraphic.GraphicInfo;  
graphicInfo0.Identity = "<InTouch:WindowName>";  
ShowGraphic( graphicInfo );
```

### Parameter

*GraphicInfo*

### Data Type

aaGraphic.GraphicInfo

### Examples

#### Show graphic within a pop-up window

```
ShowGraphic (graphicInfo);
```

#### Show an InTouch window

```
Dim graphicInfo0 as aaGraphic.GraphicInfo;  
graphicInfo0.Identity = "InTouch:Window1";  
ShowGraphic( graphicInfo0 );
```

### aaGraphic.GraphicInfo Properties

Any string properties can be a concatenation of strings and/or custom properties.

#### *Identity*

A unique name that identifies which instance has opened the graphic.

**Data Type**

String

**Additional Information**

Mandatory

The same Identity is used in the HideGraphic() script function to close the pop-up window.

**Valid Range**

The name cannot contain more than 329 characters.

The name must contain at least one letter.

Valid characters are alphanumeric and special characters (\$, #, \_).

**Example**

```
graphicInfo.Identity = "i1";
```

*GraphicName*

The name of the graphic to show.

**Data Type**

String

**Valid Range**

The name cannot contain more than 329 characters.

The name must contain at least one letter.

Valid characters are alphanumeric and special characters (\$, #, \_).

**Additional Information**

Mandatory

Browse using the **Display Galaxy Browser** or directly type the graphic name.

Galaxy name can come from:

- Graphic Toolbox, for example:  
"Symbol\_001"
- Instances, absolute or hierarchical, for example:  
"Userdefined\_001.Symbol1",  
"Userdefined\_001.Pump\_001.S1"
- Relative reference, for example:  
"Me.Symbol\_001"

If you type any invalid character or exceed the character limit, the system shows a warning message at run time. There is no validation at design time.

The graphic name can be a concatenation of constant strings and reference strings. For example: "Pump\_001" + ".Symbol\_001"; cp1 + ".Symbol\_001", where the value of cp1 = "Pump\_001"; or Obj1.Str1 + ".Symbol\_001", where the value of Obj.Str1 = "Pump\_001".

**Example**

```
graphicInfo.GraphicName = "S1";
```

*OwningObject*

The owning object of the graphic shown by the ShowGraphic() script function

**Data Type**

String

**Default Value**

Empty

**Additional Information**

Optional

Can be a concatenation of constant strings and reference strings.

Can be browsed using the **Display Automation Object Browser**, or you can type the name of the owning object.

**Example**

```
graphicInfo.OwningObject = "UserDefined_001";
```

*HasTitleBar*

Determines if the graphic is shown with a title bar.

**Data Type**

Boolean

**Default Value**

True

**Example**

```
graphicInfo.HasTitleBar = false;
```

*WindowTitle*

Specifies the title shown in the window title bar.

**Data Type**

String

**Default Value**

Empty

**Valid Range**

Limit 1024 characters

**Additional Information**

Can be a constant string, a reference, or an expression.

If you change the owning object for an AutomationObject graphic, the window title is updated accordingly.

If the *WindowTitle* parameter is empty, the value of the *Identity* parameter is shown on the title bar.



**Example**

```
graphicInfo.WindowTitle = "Graphic01";
```

*WindowType*

Specifies whether window type is modal or modeless.

**Data Type**

Enum

**Default Value**

Modeless

**Valid Range**

0, 1

**Enumerations**

WindowType	Integer
Modal	0
Modeless	1

**Examples**

```
graphicInfo.WindowType =  
aaGraphic.WindowType.<windowtype>;
```

```
graphicInfo.WindowType = 1;
```

*HasCloseButton*

Determines if the pop-up window has a close button.

**Data Type**

Boolean

**Default Value**

True

**Example**

```
graphicInfo.HasCloseButton = false;
```

*Resizable*

Determines if the pop-up window is resizable.

**Data Type**

Boolean

**Default Value**

False

**Example**

```
graphicInfo.Resizable = true;
```

*WindowLocation*

Specifies the location of the pop-up window.

**Data Type**

Enum

**Default Value**

Center

**Valid Range**

One of 0–12

**Enumerations**

<b>WindowLocation</b>	<b>Integer</b>
Center	0
Above	1
TopLeftCorner	2
Top	3
TopRightCorner	4
LeftOf	5
LeftSide	6
RightSide	7
RightOf	8
BottomLeftCorner	9
Bottom	10
BottomRightCorner	11
Below	12

**Additional Information**

If you have selected Desktop as the window relative position, Above, LeftOf, RightOf, and Below are invalid.

For more information about the behavior of the *WindowLocation* parameter, see *Chapter 16, "Working with the Show/Hide Graphics Script Functions," in the Creating and Managing ArchestrA Graphics User's Guide.*

**Examples**

```
graphicInfo.WindowLocation =
aaGraphic.WindowLocation.<WindowLocation>;

graphicInfo.WindowLocation = 1;
```

*WindowRelativePosition*

Specifies the relative position of the pop-up window.

**Data Type**

Enum

**Default Value**

Desktop

**Valid Range**

One of 0–8

**Enumerations**

WindowRelativePosition	Integer
Desktop	0
Window	1
ClientArea	2
ParentGraphic	3
ParentElement	4
Mouse	5
DesktopXY	6
WindowXY	7
ClientAreaXY	8

**Examples**

```
graphicInfo.WindowRelativePosition =
aaGraphic.WindowRelativePosition.<WindowRelativePosit
ion>;
```

```
graphicInfo.WindowRelativePosition = 1;
```

*RelativeTo*

Specifies the size of the pop-up window relative to the graphic, desktop, or customized width and height.

**Data Type**

Enum

**Default Value**

Graphic

**Valid Range**

One of 0–2

**Enumerations**

RelativeTo	Integer
Graphic	0
DeskTop	1
CustomizedWidthHeight	2

**Additional Information**

If you enter `aaGraphic.RelativeTo.CustomizedWidthHeight`, you can include the values of the height and width in the script. Otherwise, the default values are used.

**Examples**

```
graphicInfo.RelativeTo =  
aaGraphic.RelativeTo.<RelativeTo>;  
  
graphicInfo.RelativeTo = 1;
```

**X**

The horizontal position of the pop-up window.

**Data Type**

Integer

**Default Value**

0

**Valid Range**

-2,147,483,648 through 2,147,483,647

**Additional Information**

If *X* is beyond the integer range, an overflow message appears in the Logger at run time.

This parameter is applicable only if the value of the WindowRelativePosition parameter is DesktopXY, WindowXY, or ClientAreaXY.

Unlike the ShowSymbol animation, there is no boundary for this value.

**Examples**

```
graphicInfo.X = 100;
```

**Y**

Specifies the vertical position of the pop-up window.

**Data Type**

Integer

**Default Value**

0

**Valid Range**

-2,147,483,648 through 2,147,483,647

**Additional Information**

If *Y* is beyond integer range, a proper overflow message will appear in the Logger at run time.

This value is applicable only if WindowRelativePosition is DesktopXY, WindowXY, or ClientAreaXY.

Unlike the ShowSymbol animation, there is no boundary for this value.

**Examples**

```
graphicInfo.Y = 100;
```

### *Width*

Specifies the width of the pop-up window.

**Data Type**

Integer

**Default Value**

100

**Valid Range**

0–10000

**Additional Information**

Applicable only if *RelativeTo* is CustomizedWidthHeight

You can specify either the height or the width of the pop-up window. The system calculates the other, based on the aspect ratio of the symbol.

If you enter an out-of-boundary value, the system shows an “Out of range” message at run time. If the value > 10000, it is set at 10000. If the value < 0, it is set at 0.

**Examples**

```
graphicInfo.width = 500;
```

### *Height*

Specifies the height of the pop-up window.

**Data Type**

Integer

**Default Value**

100

**Valid Range**

0–10000

**Additional Information**

Applicable only if *RelativeTo* is the value of the CustomizedWidthHeight parameter.

You can specify either the height or the width of the pop-up window. The system calculates the other, based on the aspect ratio of the symbol.

If you enter an out-of-boundary value, the system shows an “Out of range” message at run time. If the value > 10000, it is set at 10000. If the value < 0, it is set at 0.

**Examples**

```
graphicInfo.height = 500;
```

### *TopMost*

Sets a value that indicates whether the ShowGraphic appears in the top most z-order window. A ShowGraphic whose Topmost property is set to true appears above all windows whose TopMost properties are set to false (same as Windows Task Manager).

**Data Type**

Boolean

**Default Value**

False

**Additional Information**

ShowGraphic windows whose Topmost properties are set to true appear above all windows whose Topmost properties are set to false. In a group of windows that have the Topmost property set to true, the active window is the topmost window.

---

**Note:** Do not create scripts that launch a non-TopMost Modal dialog from a TopMost dialog. Users will not be able to interact with the View if the Modal dialog is completely hidden by any TopMost window.

---

**Example**

```
graphicInfo.TopMost = true;
```

### *ScalePercentage*

Sets the scaling percentage of the pop-up window and the graphic it contains.

**Data Type**

Integer

**Default Value**

100

**Valid Range**

0–1000

**Additional Information**

If you enter an out-of-boundary value, the system shows an “Out of range” message at run time. If the value > 1000, it is set at 1000. If the value < 0, it is set at 0.

**Examples**

```
graphicInfo.ScalePercentage = 150;
```

### *KeepOnMonitor*

Specifies that a pop-up window should appear entirely within the boundaries of an application window.

**Data Type**

Boolean

**Default Value**

True

**Example**

```
graphicInfo.KeepOnMonitor = true;
```

*StretchGraphicToFitWindowSize*

Determines if the graphic is scaled to the current size of the pop-up window.

**Data Type**

Boolean

**Default Value**

True

**Additional Information**

Applicable only if the value of the ScalePercentage parameter is greater than 100.

**Examples**

```
graphicInfo.StretchGraphicToFitWindowSize = false;
```

*StretchWindowToScreenWidth*

Determines if the pop-up window is scaled to the same width as the screen.

**Data Type**

Boolean

**Default Value**

False

**Additional Information**

Applicable only if the WindowRelativePosition parameter is Desktop, Window, Client Area, ParentGraphic, or ParentElement.

**Examples**

```
graphicInfo.StretchWindowToScreenWidth = true;
```

*StretchWindowToScreenHeight*

Determines if the pop-up window is scaled to the same height as the screen.

**Data Type**

Boolean

**Default Value**

False

**Additional Information**

Applicable only if the WindowRelativePosition parameter is Desktop, Window, Client Area, ParentGraphic, or ParentElement.

**Examples**

```
graphicInfo.StretchWindowToScreenHeight = true;
```

*CustomProperties*

Sets the custom properties of the symbol being shown.

**Data Type**

CustomPropertyValuePair[] array

**Additional Information**

The first three parameters are custom property name, value, and IsConstant.

Both custom property and the value can be a constant string, reference, or concatenation of strings.

If the parameter IsConstant = True, the value is treated as a constant. Otherwise, the value is treated as a reference.

The array index starts at 1.

**Examples**

```
Dim cpValues [4] as aaGraphic.CustomPropertyValuePair;
cpValues[1] = new
aaGraphic.CustomPropertyValuePair("CP1", 20, true);
cpValues[2] = new
aaGraphic.CustomPropertyValuePair("CP2",
Pump.PV.TagName, true);
cpValues[3] = new
aaGraphic.CustomPropertyValuePair("CP3", "CP"+var1,
CP2 + "001" + ".Speed", true);
cpValues[4] = new
aaGraphic.CustomPropertyValuePair("CP3",
"InTouch:Tag1", false);
graphicInfo.CustomProperties = cpValues;
```

**Remarks**

Any parameter that has default value in the GraphicInfo is optional. If no input value specified for these parameters, the default values are used at run time. Any parameter except the Enum data type can be a constant, reference, or expression.

For more information, see Chapter 16, "Working with the Show/Hide Graphics Script Functions," in the *Creating and Managing ArchastrA Graphics User's Guide*.

**Examples for ShowGraphic**

Basic script example:

```
Dim graphicInfo as aaGraphic.GraphicInfo;
graphicInfo.Identity = "Script_001";
graphicInfo.GraphicName = "Symbol_001";
ShowGraphic( graphicInfo );
```

Advanced script example:

```
Dim graphicInfo as aaGraphic.GraphicInfo;
```



```

Dim cpValues [2] as aaGraphic.CustomPropertyValuePair;
cpValues[1] = new
    aaGraphic.CustomPropertyValuePair("CP1", 20, true);
cpValues[2] = new
    aaGraphic.CustomPropertyValuePair("CP2",
    "Pump.PV.TagName", false);
graphicInfo.Identity = "i1";
graphicInfo.GraphicName = "S1";
graphicInfo.OwningObject = "UserDefined_001";
graphicInfo.WindowTitle = "Graphic01";
graphicInfo.Resizable = false;
graphicInfo.CustomProperties=cpValues;
ShowGraphic( graphicInfo );

```

Where "i1" is string Identity and the symbol "S1" contains custom property CP1 and CP2.

#### See Also

HideGraphic(), HideSelf()

## SignedAlarmAck()

Acknowledges one or more alarms on ArchestrA attributes, optionally requiring a signature if any of the indicated alarms falls within a designated priority range.

This function is supported only for client scripting and not object scripting.

#### Category

Miscellaneous

#### Syntax

```

int SignedAlarmAck(String Alarm_List,
Boolean Signature_Reqd_for_Range,
Integer Min_Priority,
Integer Max_Priority,
String Default_Ack_Comment,
Boolean Ack_Comment_Is_Editable,
String TitleBar_Caption,
String Message_Caption
);

```

## Parameters

### *Alarm\_List*

The list of alarms to be acknowledged. The list must be a single text string with each alarm name separated by a space or a comma.

#### **Data Type**

String

#### **Valid Range**

Limit 1024 characters

#### **Additional Information**

Can be a constant string, a reference, or an expression.

Only alarms on ArcestrA attributes are supported.

If there is any invalid alarm in the list, then none of the alarms are acknowledged.

#### **Examples**

Example 1:

```
"UD1.analog_001.HiHi"
```

The collection is represented as a text string, with alarms separated by blanks and/or commas.

Example 2:

```
"UD1.analog_001.HiHi UD9.x14.dev.major"
```

Example 3:

```
"UD1.analog_001.HiHi, UD9.x14.dev.major"
```

Example 4, an array of strings such as:

```
Pump1.AlarmArray[1] = "Pump1.Level.HiHi"
```

```
Pump1.AlarmArray[2] = "Pump1.Level.LoLo"
```

uses the function as follows:

```
SignedAlarmAck(Pump1.AlarmArray[ ], ...)
```

The script passes to the function the following single string:

```
"Pump1.Level.HiHi, Pump1.Level.LoLo"
```

### *Signature\_Reqd\_for\_Range*

Indicates whether a signature is required for acknowledging alarms.

#### **Data Type**

Bool

#### **Additional Information**

Can be a constant, a reference, or an expression.

### *Min\_Priority*

Represents the minimum priority value of the range for which the signature is required.

**Data Type**

Integer

**Valid Range**1-999; must be less than or equal to the *Max\_Priority* value.**Additional Information**

Can be a constant, a reference, or an expression.

*Max\_Priority*

Represents the maximum priority value of the range for which the signature is required.

**Data Type**

Integer

**Valid Range**1-999; must be greater than or equal to the *Min\_Priority* value.**Additional Information**

Can be a constant, a reference, or an expression.

*Default\_Ack\_Comment*Comment to be shown in the **Acknowledge Alarms** dialog box.**Data Type**

String

**Valid Range**

Limit 200 characters

**Additional Information**

Can be a constant, a reference or an expression.

If the parameter is empty, then no default comment is shown in the **Acknowledge Alarms** dialog box.*Ack\_Comment\_Is\_Editable*

Indicates whether the run-time user can modify the acknowledgement comment.

**Data Type**

Bool

**Additional Information**

Can be a constant, a reference, or an expression.

If set to False, the **Comment** box in the **Acknowledge Alarms** dialog box is unavailable.*TitleBar\_Caption*Shows a title in the title bar of the **Acknowledge Alarms** dialog box.**Data Type**

String

**Valid Range**

Limit 1024 characters

**Additional Information**

Can be a constant, a reference, or an expression.

If the *TitleBar\_Caption* is empty, the default title, **Acknowledge Alarms**, is shown.

*Message\_Caption*

Shows a customizable message to the run-time user in the **Acknowledge Alarms** dialog box.

**Data Type**

String

**Valid Range**

Limit 250 characters

**Additional Information**

Can be a constant, a reference, or an expression.

Use the parameter to provide more information on the alarm to the run-time user.

This message is not propagated to the event record.

**Return Values**

Return values indicate success or failure status. A non-zero value indicates type of failure.

- 1 The user canceled the operation.  
The function writes a message to the Logger indicating user cancellation.
- 2 No alarms are waiting for acknowledgement.
- 0 The function is successful and the following are all true:
  - The function parameters are valid.
  - The user credentials are valid (or no credentials are needed).
  - The user did not cancel the operation.
  - Function wrote to the *.AckMsg* attributes of the indicated alarms.
- 1 The function failed due to any error that is not covered by the other specified return values.
- 2 One or more parameters were not coerced to the appropriate data type at run time.

Example: Parameter is a reference with Boolean as the expected data type. At run time, reference is to a String data type that cannot be coerced to True or False.

The function writes a message to the Logger.

- 3 The *Alarm\_List* parameter was not valid at run time.
- String was null, empty or contained no attribute references.
  - Contained one or more items that were not valid attribute references.
  - Contained one or more attribute references that did not exist or did not identify valid alarm primitives.

If *Alarm\_List* contains a mixture of valid and invalid references, the function does nothing. The function does not attempt to operate on the valid references, and returns this error status.

- 4 The *Min\_Priority* or *MaxPriority* values do not fall within the range of 1 to 999.

The function writes a message to the Logger indicating which parameter was out of range and showing the actual value.

- 5 The *Min\_Priority* value is greater than the *Max\_Priority* value.

The function writes a message to the Logger identifying the problem and showing the actual values.

---

**Note:** A return value of zero does not indicate if the alarms are acknowledged, only that the function wrote to the AckMsg attributes. The alarms may not be acknowledged due to insufficient permission or if the alarms have already been acknowledged.

---

### Remarks

For more information about using the SignedAlarmAck() function, see "Signature Security for Acknowledging Alarms" in Chapter 11, "Adding and Maintaining Symbol Scripts," in the *Creating and Managing ArchestrA Graphics User's Guide*.

### Examples

```
Dim n as Integer;
n = SignedAlarmAck("UD1.analog_001.HiHi UD9.x14.dev.major",
true, 1, 250, "Acknowledged by script", true, "Acking Tank
Alarms", "Acknowledge the tank alarms");
```

Using an array of strings:

```
dim arr[2] as String;
arr[1] = "UD1.analog_001.HiHi";
arr[2] = "UD9.x14.dev.major";
n = SignedAlarmAck(arr[], true, 200, 500, "Aked by script",
true, "Acking Tank Alarms", "Please acknowledge the tank
alarms.");
```

## SignedWrite()

Performs a write to an AutomationObject attribute that has a Secured Write or Verified Write security classification.

### Category

Miscellaneous

### Syntax

```
int SignedWrite(string Attribute,  
object Value,  
string ReasonDescription,  
Bool Comment_Is_Editable,  
Enum Comment_Enforcement,  
string[] Predefined_Comment_List  
);
```

Brackets [ ] indicate an array.

### Parameters

#### *Attribute*

The attribute to be updated.

#### **Data Type**

String

#### **Additional Information**

Can be a constant string, a reference, or an expression.

Supports bound and nested bound references.

For detailed examples of *Attribute* parameter uses, see "Examples of Using the Attribute Parameter in the SignedWrite() Function" in Chapter 3, "Managing Symbols," in the *Creating and Managing ArcestrA Graphics User's Guide*.

#### **Examples**

Example 1:

```
"UserDefined_001.temp"
```

Example 2:

```
"Pump15" + ".valve4"
```

Example 3:

With UDO\_7 containing two string attributes, namestrA and namestrB set to the values "Tank1" and "Tank5" respectively, the following script writes to Tank1.Level or Tank5.Level according to whether strselect is "A" or "B":

```
Dim strselect As String;
```

```

Dim x As Indirect;
{ logic to set strselect to "A" or "B" }
x.BindTo ("UDO_7.namestr" + strselect);
SignedWrite(x + ".Level", 243, "Set " + x + "
Level", true, 0, null);

```

*Value*

The value to be written.

**Data Type**

Object

**Valid Range**

Must match data type of the attribute being updated.

**Additional Information**

Can be a constant value, a reference, an expression, or *NULL* if nothing is to be entered.

*ReasonDescription*

Text that explains the purpose of the target attribute and the impact of changing it.

**Data Type**

String

**Valid Range**

Maximum of 256 characters.

**Additional Information**

Can be a constant string, a reference, or an expression.

The *ReasonDescription* is passed to the indicated Attribute as part of the write operation. The object also includes the user's write comment, if any. A Field Attribute description is used for the *ReasonDescription* parameter only if the attribute is a Field Attribute and it has a description (is not null). Otherwise, the Short Description for the corresponding ApplicationObject is used for the *ReasonDescription* parameter.

*Comment\_is \_Editable*

Indicates whether user can edit the write comment.

**Data Type**

Bool

**Additional Information**

Can be a constant value, a reference, or an expression.

If set to True: The comment text box is enabled with exceptions. If *Comment\_Is\_Editable* is true and if the *Comment\_Enforcement* parameter is PredefinedOnly, the comment text box is disabled. At run time, the user can only select a comment from the predefined comment list.

If the *Comment\_Enforcement* parameter is not `PredefinedOnly`, the comment list and box are enabled. You can select a comment from the comment list and modify it in the comment box.

If the predefined list is empty, the comment list is not shown in the dialog box.

If set to `False`: The predefined comment list does not appear in the Secured Write or Verified Write dialog boxes. The editable comment text box is disabled.

#### *Comment\_Enforcement*

Contains choices of `Optional`, `Mandatory` and `PredefinedOnly`.

##### **Data Type**

Enum

##### **Enumerations**

`Optional = 0`

The run-time user can enter a comment or leave it blank.

`Mandatory = 1`

The run-time user must add a comment, either by selecting from the comment list or by entering a comment in the comment box.

`PredefinedOnly = 2`

The run-time user can select a comment from the comment list only. The comment text box is disabled.

##### **Additional Information**

Can be a constant, a reference, or an expression.

#### *Predefined\_Comment\_List*

An array of strings that can be used as predefined comments.

##### **Data Type**

`String[]`

##### **Valid Range**

Maximum of 20 comments, each with a maximum of 200 characters.

##### **Additional Information**

The array can be empty (number of elements is 0).

Can be a constant, a reference, an expression, or `NULL` if empty.

Can reference an attribute that contains an array of strings.

If no predefined comment is entered, the predefined comment list is disabled at run time.

If *Comment\_Is\_Editable* is `False`, the predefined comment is still placed in the editable comment text box, but the user cannot modify it at run time.



## Return Values

Return values indicate success or failure status. A non-zero value indicates type of failure.

- 0 The function returns a value of 0 (meaning success) if the following are all true:
  - The function parameters were valid.
  - The write operation was successfully placed on the queue for Secured and Verified Writes.
  - If the user cancels the operation, a message is written to the Logger indicating user cancellation.
- 1 The function failed due to any error that is not covered by the other specified return values. This includes any error that is not covered by the other specified return values. If there is a failure, a specific message is logged in the Logger.
- 2 One or more parameters were not coerced to the appropriate data type at run time.

Example: Parameter is a reference with Boolean as the expected data type. At run time, reference is to a String data type that cannot be coerced to True or False. The function returns this value and writes a message to the Logger.
- 3 The attribute parameter was not valid at run time.
  - Attribute string was null, empty, or contained no attribute reference.
  - Attribute string contained an item that was not a valid attribute reference.
  - Attribute string contained an attribute reference that did not exist.
  - Attribute string contained an attribute reference that was not of the Secured Write or Verified Write security classification.

The function writes a message to the Logger identifying the error and the invalid attribute string.
- 4 The *Comment\_Enforcement* parameter value was out of the range of valid enumerators.

### Remarks

The SignedWrite() function is supported only for client scripting and not for object scripting.

A return value of 0 does not indicate whether the attribute was updated, only that the function placed an entry on the queue to write to the attribute. The operator may decide to cancel the operation after the Secured Write or Verified write dialog box is presented. In this case the attribute is not updated and a message is placed in the Logger indicating that the user canceled the operation. Even if the user enters valid credentials and clicks **OK**, the attribute still might not have been updated because of inadequate permission or data coercion problems.

The SignedWrite() function supports the custom property passed as the first parameter with opened and closed quotation marks, “”.

If you configure the custom property CP as shown in the following script, the function attempts to resolve CP and determine if it has a reference. If it has a reference, then the reference is retrieved and the write is performed on the reference.

```
SignedWrite("CP", value, reason, editable, enforcement, null);
```

For more information about using the SignedWrite() function, see "Working with the SignedWrite() Function for Secured and Verified Writes" in Chapter 3, "Managing Symbols," in the *Creating and Managing ArcestrA Graphics User's Guide*.

### Examples

```
SignedWrite ("UserDefined_001.temp", 185, "This will change the  
oven temperature", true, 1, null);
```

The following example shows the user an array of predefined comments:

```
Dim n as Integer;  
  
n = SignedWrite("UserDefined_001.temp", 185, "This will change  
the oven temperature", true, 1,  
UserDefined_001.OvenCommentArray[ ]);
```

where UserDefined\_001.OvenCommentArray is an attribute containing an array of strings.

## Sin()

Returns the sine of an angle in degrees.

### Category

Math

### Syntax

```
Result = Sin( Number );
```

### Parameter

*Number*

Angle in degrees. Any number or numeric attribute.

### Examples

```
Sin(90); ' returns 1;
```

```
Sin(0); ' returns 0;
```

This example shows how to use the function in a math expression:

```
wave = 100 * Sin (6 * Now().Second);
```

### See Also

Cos(), Tan(), ArcCos(), ArcSin(), ArcTan()

## Sqrt()

Returns the square root of a number.

### Category

Math

### Syntax

```
RealResult = Sqrt( Number );
```

### Parameter

*Number*

Any number or numeric attribute.

### Example

This example takes the value of me.PV and returns the square root as the value of x:

```
x=Sqrt (me.PV);
```

## StringASCII()

Returns the ASCII value of the first character in a specified string.

### Category

String

### Syntax

```
IntegerResult = StringASCII( Char );
```

### Parameter

*Char*

Alphanumeric character or string or string attribute.

### Remarks

When this function is processed, only the single character is tested or affected. If the string provided to StringASCII contains more than one character, only the first character of the string is tested.

### Examples

```
StringASCII("A"); ' returns 65;
```

```
StringASCII("A Mixer is Running"); ' returns 65;
```

```
StringASCII("a mixer is running"); ' returns 97;
```

### See Also

StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringChar()

Returns the character corresponding to a specified ASCII code.

### Category

String

### Syntax

```
StringResult = StringChar( ASCII );
```

### Parameter

*ASCII*

ASCII code or an integer attribute.

**Remarks**

Use the `StringChar` function to add ASCII characters not normally represented on the keyboard to a string attribute.

This function is also useful for SQL commands. The where expression sometimes requires double quotation marks around string values, so use `StringChar(34)`.

**Example**

In this example, a [Carriage Return (13)] and [Line Feed (10)] are added to the end of `StringAttribute` and passed to `ControlString`. Inserting characters out of the normal 32-126 range of displayable ASCII characters can be very useful for creating control codes for external devices such as printers or modems.

```
ControlString = StringAttribute+StringChar(13)+StringChar(10);
```

## StringCompare()

Compares a string value with another string.

**Category**

String

**Syntax**

```
StringCompare( Text1, Text2 );
```

**Parameters**

*Text1*

First string in the comparison.

*Text2*

Second string in the comparison.

**Return Value**

The return value is zero if the strings are identical, -1 if `Text1`'s value is less than `Text2`, or 1 if `Text1`'s value is greater than `Text2`.

**Example**

```
Result = StringCompare ("Text1","Text2"); (or)
```

```
Result = StringCompare (MText1,MText2);
```

Where `Result` is an Integer or Real tag and `MText1` and `MText2` are Memory Message tags.

**See Also**

StringASCII(), StringChar(), StringFromReal(), StringFromTime(), StringFromTimeLocal(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringCompareNoCase()

Compares a string value with another string and ignores the case.

**Category**

String

**Syntax**

```
SStringCompareNoCase( Text1, Text2 );
```

**Parameters**

*Text1*

First string in the comparison.

*Text2*

Second string in the comparison.

**Return Value**

The return value is zero if the strings are identical (ignoring case), -1 if Text1's value is less than Text2 (ignoring case), or 1 if Text1's value is greater than Text2 (ignoring case).

**Example**

```
Result = StringCompareNoCase ("Text1","TEXT1"); (or)
```

```
Result = StringCompareNoCase (MText1,MText2);
```

Where Result is an Integer or Real tag and MText1 and MText2 are Memory Message tags.

**See Also**

StringASCII(), StringChar(), StringFromReal(), StringFromTime(), StringFromTimeLocal(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringFromGMTTimeToLocal()

Converts a time value (in seconds since Jan-01-1970) to a particular string representation. This is the same as StringFromTime().

**Category**

String

**Syntax**

```
MessageResult=StringFromGMTTimeToLocal (SecsSince1-1-70, StringType);
```

**Parameters***SecsSince1-1-70*

Is converted to the StringType specified and the result is stored in MessageResult.

*StringType*

Determines the display method:

1 = Displays the date in the same format set from the windows control Panel. (Similar to that displayed for \$DateString.)

2 = Displays the time in the same format set from the Windows control Panel. (Similar to that displayed for \$TimeString.)

3 = Displays a 24-character string indicating both the date and time: "Wed Jan 02 02:03:55 1993"

4 = Displays the short form for the day of the week: "Wed"

5 = Displays the long form for the day of the week: "Wednesday"

**Remarks**

Any adjustments necessary due to Daylight Savings Time are automatically applied to the return result. Therefore, it is not necessary to make any manual adjustments to the input value to convert to DST.

**Example**

This example assumes that the time zone on the local node is Pacific Standard Time (UTC-0800). The UTC time passed to the function is 12:00:00 AM on Friday, 1/2/1970. Since PST is 8 hours behind UTC, the function will return the following results:

```
StringFromGMTTimeToLocal (86400, 1); ' returns "1/1/1970"
StringFromGMTTimeToLocal (86400, 2); ' returns "04:00:00 PM"
StringFromGMTTimeToLocal (86400, 3); ' returns "Thu Jan 01
    16:00:00 1970"
StringFromGMTTimeToLocal (86400, 4); ' returns "Thu"
StringFromGMTTimeToLocal (86400, 5); ' returns "Thursday"
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringFromTimeLocal(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringFromIntg()

Converts an integer value into its string representation in another base and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringFromIntg( Number, numberBase );
```

**Parameters**

*Number*

Number to convert. Any number or an integer attribute.

*numberBase*

Base to use in conversion. Any number or an integer attribute.

**Examples**

```
StringFromIntg(26, 2); ' returns "11010"
```

```
StringFromIntg(26, 8); ' returns "32"
```

```
StringFromIntg(26, 16); ' returns "1A"
```

**See Also**

StringASCII(), StringChar(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringFromReal()

Converts a real value into its string representation, either as a floating-point number or in exponential notation, and returns the result.

**Category**

String



**Syntax**

```
StringResult = StringFromReal( Number, Precision, Type );
```

**Parameters***Number*

Converted to the *Precision* and *Type* specified. Any number or a float attribute.

*Precision*

Specifies how many decimal places is shown. Any number or an integer attribute.

*Type*

A string value that determines the display method. Possible values are:

f = Display in floating-point notation.

e = Display in exponential notation with a lowercase "e."

E = Display in exponential notation with an uppercase "E" followed by a plus sign and at least three exponential digits.

**Examples**

```
StringFromReal(263.355, 2, "f"); ' returns "263.36";
```

```
StringFromReal(263.355, 2, "e"); ' returns "2.63e2";
```

```
StringFromReal(263.355, 2, "E"); ' returns "2.63 E+002";
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringFromTime()

Converts a time value (in seconds since January 1, 1970) into a particular string representation and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringFromTime( SecsSince1-1-70, StringType );
```

**Parameters***SecsSince1-1-70*

Converted to the *StringType* specified.

*StringType*

Determines the display method. Possible values are:

1 = Shows the date in the same format set from the Windows Control Panel.

2 = Shows the time in the same format set from the Windows Control Panel.

3 = Shows a 24-character string indicating both the date and time: "Wed Jan 02 02:03:55 1993"

4 = Shows the short form for a day of the week: "Wed"

5 = Shows the long form for a day of the week: "Wednesday"

**Remarks**

The time value is UTC equivalent: number of elapsed seconds since January 1, 1970 GMT. The value returned reflects the local time.

**Examples**

```
StringFromTime(86400, 1); ' returns "1/2/1970"
```

```
StringFromTime(86400, 2); ' returns "12:00:00 AM"
```

```
StringFromTime(86400, 3); ' returns "Fri Jan 02 00:00:00 1970"
```

```
StringFromTime(86400, 4); ' returns "Fri"
```

```
StringFromTime(86400, 5); ' returns "Friday"
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringFromTimeLocal()

Converts a time value (in seconds since Jan-01-1970) into a particular string representation. The value returned also represents local time.

**Category**

String

**Syntax**

```
MessageResult=StringFromTimeLocal (SecsSince1-1-70,
```

StringType);

### Parameters

#### *SecsSince1-1-70*

Is converted to the StringType specified and the result is stored in MessageResult.

#### *StringType*

Determines the display method:

1 = Displays the date in the same format set from the windows control Panel. (Similar to that displayed for \$DateString.)

2 = Displays the time in the same format set from the Windows control Panel. (Similar to that displayed for \$TimeString.)

3 = Displays a 24-character string indicating both the date and time: "Wed Jan 02 02:03:55 1993"

4 = Displays the short form for the day of the week: "Wed"

5 = Displays the long form for the day of the week: "Wednesday"

### Remarks

Any adjustments necessary due to Daylight Savings Time will automatically be applied to the return result. Therefore, it is not necessary to make any manual adjustments for DST to the input value.

### Example

```
StringFromTimeLocal (86400, 1); ' returns "1/2/1970"
StringFromTimeLocal (86400, 2); ' returns "12:00:00 AM"
StringFromTimeLocal (86400, 3); ' returns "Fri Jan 02 00:00:00
1970"
StringFromTimeLocal (86400, 4); ' returns "Fri"
StringFromTimeLocal (86400, 5); ' returns "Friday"
```

### See Also

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringInString()

Returns the position in a string of text where a specified string first occurs.

**Category**

String

**Syntax**

```
IntegerResult = StringInString( Text, SearchFor, StartPos,  
    CaseSens );
```

**Parameters***Text*

The string that is searched. Actual string or a string attribute.

*SearchFor*

The string to be searched for. Actual string or a string attribute.

*StartPos*

Determines the position in the text where the search begins. Any number or an integer attribute.

*CaseSens*

Determines whether the search is case-sensitive.

0 = Not case-sensitive

1 = Case-sensitive

Any number or an integer attribute.

**Remarks**

If multiple occurrences of *SearchFor* are found, the location of the first is returned.

**Examples**

```
StringInString("The mixer is running", "mix", 1, 0) '  
    returns 5;
```

```
StringInString("Today is Thursday", "day", 1, 0) ' returns  
    3;
```

```
StringInString("Today is Thursday", "day", 10, 0) '  
    returns 15;
```

```
StringInString("Today is Veteran's Day", "Day", 1, 1) '  
    returns 20;
```

```
StringInString("Today is Veteran's Day", "Night", 1, 1) '  
    returns 0;
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringLeft()

Returns a specified number of characters in a string value, starting with the leftmost string character.

### Category

String

### Syntax

```
StringResult = StringLeft( Text, Chars );
```

### Parameters

*Text*

Actual string or a string attribute.

*Chars*

Number of characters to return or an integer attribute.

### Remarks

If *Chars* is set to 0, the entire string is returned.

### Examples

```
StringLeft("The Control Pump is On", 3) ' returns "The";  
StringLeft("Pump 01 is On", 4) ' returns "Pump";  
StringLeft("Pump 01 is On", 96) ' returns "Pump 01 is On";  
StringLeft("The Control Pump is On", 0) ' returns "The  
Control Pump is On";
```

### See Also

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringLen()

Returns the number of characters in a string.

### Category

String

### Syntax

```
IntegerResult = StringLen( Text );
```

**Parameter***Text*

Actual string or a string attribute.

**Remarks**

All the characters in the string attribute are counted, including blank spaces and those not normally shown on the screen.

**Examples**

```
StringLen("Twelve percent") ' returns 14;
```

```
StringLen("12%") ' returns 3;
```

```
StringLen("The end." + StringChar(13)) ' returns 9;
```

The carriage return character is ASCII 13.

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringLower()

Converts all uppercase characters in text string to lowercase and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringLower( Text );
```

**Parameter***Text*

String to be converted to lowercase. Actual string or a string attribute.

**Remarks**

Lowercase characters, symbols, numbers, and other special characters are not affected.

**Examples**

```
StringLower("TURBINE") ' returns "turbine";
```

```
StringLower("22.2 Is The Value") ' returns "22.2 is the value";
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringMid()

Extracts a specific number of characters from a starting point within a string and returns the extracted character string as the result.

**Category**

String

**Syntax**

```
StringResult = StringMid( Text, StartChar, Chars );
```

**Parameters***Text*

Actual string or a string attribute to extract a range of characters.

*StartChar*

The position of the first character within the string to extract. Any number or an integer attribute.

*Chars*

The number of characters within the string to return. Any number or an integer attribute.

**Remarks**

This function is slightly different than the StringLeft() function and StringRight() function in that it allows you to specify both the start and end of the string that is to be extracted.

**Examples**

```
StringMid("The Furnace is Overheating",5,7); ' returns  
"Furnace";
```

```
StringMid("The Furnace is Overheating",13,3); ' returns  
"is ";
```

```
StringMid("The Furnace is Overheating",16,50); ' returns  
"Overheating"
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringReplace()

Replaces or changes specific parts of a provided string and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringReplace( Text, SearchFor, ReplaceWith,  
                             CaseSens, NumToReplace, MatchWholeWords );
```

**Parameters***Text*

The string in which characters, words, or phrases will be replaced. Actual string or a string attribute.

*SearchFor*

The string to search for and replace. Actual string or a string attribute.

*ReplaceWith*

The replacement string. Actual string or a string attribute.

*CaseSens*

Determines whether the search is case-sensitive. (0=no and 1=yes) Any number or an integer attribute.

*NumToReplace*

Determines the number of occurrences to replace. Any number or an integer attribute. To indicate all occurrences, set this value to 1.

*MatchWholeWords*

Determines whether the function limits its replacement to whole words. (0=no and 1=yes) Any number or an integer attribute. If *MatchWholeWords* is turned on (set to 1) and the *SearchFor* is set to "and", the "and" in "handle" are not replaced. If the *MatchWholeWords* is turned off (set to 0), it is replaced.

**Remarks**

Use this function to replace characters, words, or phrases within a string.



The `StringReplace()` function does not recognize special characters, such as `@ # $ % & * ( )`. It reads them as delimiters. For example, if the function `StringReplace() (abc#,abc#,1234,0,1,1)` is processed, there is no replacement. The `#` sign is read as a delimiter instead of a character.

### Examples

```
StringReplace("In From Within","In","Out",0,1,0) ' returns "Out
From Within" (replaces only the first one);
```

```
StringReplace("In From Within","In","Out",0,-1,0) ' returns
"Out From without" (replaces all occurrences);
```

```
StringReplace("In From Within","In","Out",1,-1,0) ' returns
"Out From Within" (replaces all that match case);
```

```
StringReplace("In From Within","In","Out",0,-1,1) ' returns
"Out From Within" (replaces all that are whole words);
```

### See Also

`StringASCII()`, `StringChar()`, `StringFromIntg()`, `StringFromReal()`, `StringFromTime()`, `StringInString()`, `StringLeft()`, `StringLen()`, `StringLower()`, `StringMid()`, `StringRight()`, `StringSpace()`, `StringTest()`, `StringToIntg()`, `StringToReal()`, `StringTrim()`, `StringUpper()`, `Text()`

## StringRight()

Returns the specified number of characters starting at the right-most character of text.

### Category

String

### Syntax

```
StringResult = StringRight( Text, Chars );
```

### Parameters

*Text*

Actual string or a string attribute.

*Chars*

The number of characters to return or an integer attribute.

### Remarks

If *Chars* is set to 0, the entire string is returned.

### Examples

```
StringRight("The Pump is On", 2) ' returns "On";
```

```
StringRight("The Pump is On", 5) ' returns "is On";
```

```
StringRight("The Pump is On", 87) ' returns "The Pump is
On";

StringRight("The Pump is On", 0) ' returns "The Pump is
On";
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringSpace()

Generates a string of spaces either within a string attribute or within an expression and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringSpace( NumSpaces );
```

**Parameter**

*NumSpaces*

Number of spaces to return. Any number or an integer attribute.

**Examples**

All spaces are represented by the "x" character:

```
StringSpace(4) ' returns "xxxx";

"Pump" + StringSpace(1) + "Station" ' returns
"PumpxStation";
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), StringUpper(), Text()

## StringTest()

Tests the first character of text to determine whether it is of a certain type and returns the result.

**Category**

String

**Syntax**

```
DiscreteResult = StringTest( Text, TestType );
```

**Parameters***Text*

String that function acts on. Actual string or a string attribute.

*TestType*

Determines the type of test. Possible values are:

1 = Alphanumeric character ('A-Z', 'a-z' and '0-9')

2 = Numeric character ('0-9')

3 = Alphabetic character ('A-Z' and 'a-z')

4 = Uppercase character ('A-Z')

5 = Lowercase character ('a-'z')

6 = Punctuation character (0x21-0x2F)

7 = ASCII characters (0x00 - 0x7F)

8 = Hexadecimal characters ('A-F' or 'a-f' or '0-9')

9 = Printable character (0x20-0x7E)

10 = Control character (0x00-0x1F or 0x7F)

11 = White Space characters (0x09-0x0D or 0x20)

**Remarks**

*StringTest()* function returns true to *DiscreteResult* if the first character in *Text* is of the type specified by *TestType*. Otherwise, false is returned. If the *StringTest()* function contains more than one character, only the first character of the attribute is tested.

**Examples**

```
StringTest("ACB123",1) ' returns 1;
```

```
StringTest("ABC123",5) ' returns 0;
```

**See Also**

*StringASCII()*, *StringChar()*, *StringFromIntg()*, *StringFromReal()*, *StringFromTime()*, *StringInString()*, *StringLeft()*, *StringLen()*, *StringLower()*, *StringMid()*, *StringReplace()*, *StringRight()*, *StringSpace()*, *StringToIntg()*, *StringToReal()*, *StringTrim()*, *StringUpper()*, *Text()*

## StringToIntg()

Converts the numeric value of a string to an integer value and returns the result.

### Category

String

### Syntax

```
IntegerResult = StringToIntg( Text );
```

### Parameter

*Text*

String that function acts on. Actual string or a string attribute.

### Remarks

When this statement is evaluated, the system reads the first character of the string for a numeric value. If the first character is other than a number, the string's value is equated to zero (0). Blank spaces are ignored. If the first character is a number, the system continues to read the subsequent characters until a non-numeric value is detected.

### Examples

```
StringToIntg("ABCD"); ' returns 0;
```

```
StringToIntg("22.2 is the Value"); ' returns 22 (since  
integers are whole numbers);
```

```
StringToIntg("The Value is 22"); ' returns 0;
```

### See Also

[StringASCII\(\)](#), [StringChar\(\)](#), [StringFromIntg\(\)](#), [StringFromReal\(\)](#),  
[StringFromTime\(\)](#), [StringInString\(\)](#), [StringLeft\(\)](#), [StringLen\(\)](#),  
[StringLower\(\)](#), [StringMid\(\)](#), [StringReplace\(\)](#), [StringRight\(\)](#),  
[StringSpace\(\)](#), [StringTest\(\)](#), [StringToReal\(\)](#), [StringTrim\(\)](#),  
[StringUpper\(\)](#), [Text\(\)](#)

## StringToReal()

Converts the numeric value of a string to a real (floating point) value and returns the result.

### Category

String

### Syntax

```
RealResult = StringToReal( Text );
```

**Parameter***Text*

String that function acts on. Actual string or a string attribute.

**Remarks**

When this statement is evaluated, the system reads the first character of the string for a numeric value. If the first character is other than a number (blank spaces are ignored), the string's value is equated to zero (0). If the first character is found to be a number, the system continues to read the subsequent characters until a non-numeric value is encountered.

**Examples**

```
StringToReal("ABCD"); ' returns 0;
```

```
StringToReal("22.261 is the value"); ' returns 22.261;
```

```
StringToReal("The Value is 2"); ' returns 0;
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringTrim(), StringUpper(), Text()

## StringTrim()

Removes unwanted spaces from text and returns the result.

**Category**

String

**Syntax**

```
StringResult = StringTrim( Text, TrimType );
```

**Parameter***Text*

String that is trimmed of spaces. Actual string or a string attribute.

*TrimType*

Determines how the string is trimmed. Possible values are:

1 = Remove leading spaces to the left of the first non-space character

2 = Remove trailing spaces to the right of the last non-space character

3 = Remove all spaces except for single spaces between words

### Remarks

The text is searched for white-spaces (ASCII 0x09-0x0D or 0x20) that are to be removed. *TrimType* determines the method used by the function:

### Examples

All spaces are represented by the "x" character.

```
StringTrim("xxxxxThisxisxaxxtestxxxxx", 1) ' returns  
  "Thisxisxaxxtestxxxxx";
```

```
StringTrim("xxxxxThisxisxaxxtestxxxxx", 2) ' returns  
  "xxxxxThisxisxaxxtest";
```

```
StringTrim("xxxxxThisxisxaxxtestxxxxx", 3) ' returns  
  "Thisxisxaxtest";
```

The `StringReplace()` function can remove ALL spaces from a specified a string attribute. Simply replace all the space characters with a "null."

### See Also

`StringASCII()`, `StringChar()`, `StringFromIntg()`, `StringFromReal()`,  
`StringFromTime()`, `StringInString()`, `StringLeft()`, `StringLen()`,  
`StringLower()`, `StringMid()`, `StringReplace()`, `StringRight()`,  
`StringSpace()`, `StringTest()`, `StringToIntg()`, `StringToReal()`,  
`StringUpper()`, `Text()`

## StringUpper()

Converts all lowercase text characters to uppercase and returns the result.

### Category

String

### Syntax

```
StringResult = StringUpper( Text );
```

### Parameter

*Text*

String to be converted to uppercase. Actual string or a string attribute.

### Remarks

Uppercase characters, symbols, numbers, and other special characters are not affected.

**Examples**

```
StringUpper("abcd"); ' returns "ABCD";
StringUpper("22.2 is the value"); ' returns "22.2 IS THE
    VALUE";
```

**See Also**

StringASCII(), StringChar(), StringFromIntg(), StringFromReal(), StringFromTime(), StringInString(), StringLeft(), StringLen(), StringLower(), StringMid(), StringReplace(), StringRight(), StringSpace(), StringTest(), StringToIntg(), StringToReal(), StringTrim(), Text()

## Tan()

Returns the tangent of an angle given in degrees.

**Category**

Math

**Syntax**

```
Result = Tan( Number );
```

**Parameter**

*Number*

The angle in degrees. Any number or numeric attribute.

**Examples**

```
Tan(45); ' returns 1;
Tan(0); ' returns 0;
```

This example shows how to use the function in a math expression:

```
Wave = 10 + 50 * Tan(6 * Now().Second);
```

**See Also**

Cos(), Sin(), ArcCos(), ArcSin(), ArcTan()

## Text()

Converts a number to text based on a specified format.

**Category**

String

**Syntax**

```
StringResult = Text( Number, Format );
```

**Parameters***Number*

Any number or numeric attribute.

*Format*

Format to use in conversion. Actual string or a string attribute.

**Examples**

```
Text(66, "#.00"); ' returns 66.00;
```

```
Text(22.269, "#.00"); ' returns 22.27;
```

```
Text(9.999, "#.00"); ' returns 10.00;
```

The following example shows how to use this function within another function:

```
LogMessage("The current value of FreezerRoomTemp is:" + Text  
(FreezerRoomTemp, "#.#"));
```

In the following example, MessageTag is set to "One=1 Two=2".

```
MessageTag = "One + " + Text(1, "#") + StringChar(32) + "Two +"  
+ Text(2, "#");
```

**See Also**

StringFromIntg(), StringToIntg(), StringFromReal(), StringToReal()

## Trunc()

Truncates a real (floating point) number by simply eliminating the portion to the right of the decimal point, including the decimal point, and returns the result.

**Category**

Math

**Syntax**

```
NumericResult = Trunc( Number );
```

**Parameter***Number*

Any number or numeric attribute.

**Remarks**

This function accomplishes the same result as placing the contents of a float type attribute into an integer type attribute.



**Examples**

```
Trunc(4.3); ' returns 4;
```

```
Trunc(-4.3); ' returns -4;
```

**See Also**

Round()

## WriteStatus()

Returns the enumerated write status of the last write to the specified attribute.

**Category**

Miscellaneous

**Syntax**

```
Result = WriteStatus( Attribute );
```

**Parameter**

*Attribute*

The attribute for which you want to return write status.

**Return Value**

The return statuses are:

- MxStatusOk
- MxStatusPending
- MxStatusWarning
- MxStatusCommunicationError
- MxStatusConfigurationError
- MxStatusOperationalError
- MxStatusSecurityError
- MxStatusSoftwareError
- MxStatusOtherError

**Remarks**

If the attribute has never been written to, this function returns MxStatusOk. This function always returns MxStatusOk for attributes that do not support a calculated (non-Good) quality.

**Example**

```
WriteStatus(TIC101.SP);
```

## WWControl()

Restores, minimizes, maximizes, or closes an application.

### Category

Miscellaneous

### Syntax

```
WWControl( AppTitle, ControlType );
```

### Parameters

#### *AppTitle*

The name of the application title to be controlled. Actual string or a string attribute.

#### *ControlType*

Determines how the application is controlled. Possible values are shown below. These actions are identical to clicking on their corresponding selections in the application's Control Menu. Actual string or a string attribute.

"Restore" = Activates and shows the application's window.

"Minimize" = Activates a window and shows it as an icon.

"Maximize" = Activates and shows the application's window.

"Close" = Closes an application.

### Example

```
WWControl("Calculator", "Restore");
```

### See Also

[ActivateApp\(\)](#)

## WWExecute()

Using the DDE protocol, executes a command to a specified application and topic and returns the status.

### Category

WWDDE

### Syntax

```
Status = WWExecute( Application, Topic, Command );
```

## Parameters

### *Application*

The application to which you want to send an execute command. Actual string or a string attribute.

### *Topic*

The topic within the application. Actual string or a string attribute.

### *Command*

The command to send. Actual string or a string attribute.

## Return Value

*Status* is an Integer attribute to which 1, -1, or 0 is written. The `WVExecute()` function returns 1 if the application is running, the topic exists, and the command was sent successfully. It returns 0 when the application is busy, and -1 when there is an error.

## Remarks

---

**Note:** The three WVDDE functions `Execute()`, `Poke()` and `Request()` exist for legacy purposes.

---

The *Command* string is sent to a specified application and topic.

---

**Important:** The following applies to using `WVExecute()` in synchronous scripts:

1. Never loop them (call them over and over).
  2. Never call several of them in a row and in the same script.
  3. Never use them to call a lengthy task in another DDE application.
- All three actions, though, are appropriate in asynchronous scripts.
- 

## Examples

The following statement executes a macro in Excel:

```
Macro="Macro1!TestMacro";

Command="[Run(" + StringChar(34) + Macro + StringChar(34)
+ ",0)]";

WVExecute("excel","system",Command);
```

When `WVExecute("excel","system",Command);` is processed, the following is sent to Excel (and *TestMacro* runs):

```
[Run("Macro1!TestMacro")];
```

The following script executes a macro in Microsoft Access:

```
WVExecute("MSAccess","system","MyMacro");
```

## WWPoke()

Using the DDE protocol, pokes a value to a specified application, topic, and item and returns the status.

### Category

WWDDE

### Syntax

```
Status = WWPoke( Application, Topic, Item, TextValue );
```

### Parameters

#### *Application*

The application to which you want to send the Poke command. Actual string or a string attribute.

#### *Topic*

The topic within the application. Actual string or a string attribute.

#### *Item*

The item to poke within the topic. Actual string or a string attribute.

#### *TextValue*

The value to poke. If the value you want to send is a number, you can convert it using the Text(), StringFromIntg(), or StringFromReal() functions. Actual string or a string attribute.

### Return Value

*Status* is an Integer attribute to which 1, -1, or 0 is written. The WWPoke() function returns 1 if the application is running, the topic and item exist, and the value was sent successfully. It returns 0 if the application is busy, and -1 if there is an error.

### Remarks

---

**Note:** The three WWDDE functions Execute(), Poke() and Request() exist for legacy purposes.

---

The value *TextValue* is sent to the particular application, topic, and item specified.

---

**Important:** The following applies to using WWRequest() in synchronous scripts:

1. Never loop them (call them over and over).
  2. Never call several of them in a row and in the same script.
  3. Never use them to call a lengthy task in another DDE application. All three actions, though, are appropriate in asynchronous scripts.
-

**Example**

The following statement converts a value to text and pokes the result to an Excel spreadsheet cell:

```
String=Text(Value,"0");
WWPoke("excel","[Book1.xls]sheet1","r1c1",String);
```

The behavior for WWPoke() from within the application "View" to "View" is undefined and is not supported. The WWPoke() command is not guaranteed to succeed in this instance, and the command will probably time-out without the desired results.

**See Also**

Text(), StringFromIntg(), StringFromReal()

## WWRequest()

Using the DDE protocol, makes a one-time request for a value from a particular application, topic, and item and returns the status.

**Category**

WWDDE

**Syntax**

```
Status = WWRequest( Application, Topic, Item, Attribute );
```

**Parameters***Application*

The application from which you want to request data. Actual string or a string attribute.

*Topic*

The topic within the application. Actual string or a string attribute.

*Item*

The item within the topic. Actual string or a string attribute.

*Attribute*

A string attribute, enclosed in quotation marks, that contains the requested value from the application, topic, and item. Actual string or a string attribute.

**Return Value**

*Status* is an integer attribute to which 1, -1, or 0 is written. The WWRequest() function returns 1 if the application is running, the topic and item exist, and the value was returned successfully. It returns 0 if the application is busy, and -1 if there is an error.

### Remarks

---

**Note:** The three WWDDE functions `Execute()`, `Poke()` and `Request()` exist for legacy purposes.

---

The DDE value in the particular application, topic, and item is returned into *Attribute*.

The value is returned as a string into a string attribute. If the value is a number, you can then convert it using the `StringToIntg()` or `StringToReal()` functions.

---

**Important:** Never do the following when using `WWRequest()` in synchronous scripts:

1. Loop scripts (call them over and over).
  2. Call several of scripts in a row and in the same script.
  3. Use scripts to call a lengthy task in another DDE application.
- All three actions can be done in asynchronous scripts.
- 

### Example

The following statement requests a value from an Excel spreadsheet cell and converts the resulting string into a value:

```
WWRequest("excel", "[Book1.xls]sheet1", "r1c1", Result);  
Value=StringToReal(Result);
```

### See Also

`StringToIntg()`, `StringToReal()`

## WWStringFromTime()

Converts a time value given in local time into UTC time (Coordinated Universal Time), and displays the result as a string.

### Category

String

### Syntax

```
MessageResult = wwStringFromTime(SecsSince1-1-70,  
StringType);
```

### Parameters

*SecsSince1-1-70*

Integer Type. Number of Seconds elapsed since Jan 01 00:00:00 1970.

*StringType*

Determines the display method:

1 = Displays the date in the same format set from the windows control Panel. (Similar to that displayed for \$DateString.)

2 = Displays the time in the same format set from the Windows control Panel. (Similar to that displayed for \$TimeString.)

3 = Displays a 24-character string indicating both the date and time: "Wed Jan 02 02:03:55 1993"

4 = Displays the short form for the day of the week: "Wed"

5 = Displays the long form for the day of the week: "Wednesday"

**Remarks**

Any adjustments necessary due to Daylight Savings Time will automatically be applied to the return result. Therefore, it is not necessary to make any manual adjustments for DST to the input value.

**Example**

This example assumes that the time zone on the local node is Pacific Standard Time (UTC-0800). The local time passed to the function is 04:00:00 PM on Thursday, 1/1/1970. Since PST is 8 hours behind UTC, the function will return the following results:

```
wwStringFromTime(57600, 1) will return "1/2/70"
```

```
wwStringFromTime(57600, 2) will return "12:00:00 AM"
```

```
wwStringFromTime(57600, 3) will return "Fri Jan 02 00:00:00
1970"
```

```
wwStringFromTime(57600, 4) will return "Fri"
```

```
wwStringFromTime(57600, 5) will return "Friday"
```

## QuickScript .NET Variables

QuickScript .NET variables must be declared before they can be used in QuickScript .NET scripts. Variables can be used on both the left and right side of statements and expressions.

Local variables or attributes can be used together in the same script. Variables declared within the script body lose their value after the script is executed. Those declared in the script body cannot be accessed by other scripts.

Variables declared in the **Declarations** area maintain their values throughout the lifetime of the object that the script is associated with.

Each variable must be declared in the script by a separate DIM statement followed by a semicolon. Enter DIM statements in the **Declarations** area of the **Script** tab page. The DIM statement syntax is as follows:

```
DIM <variable_name> [ ( <upper_bound>
    [, <upper_bound >[, < upper_bound >]] ) ]
    [ AS <data_type> ];
```

where:

DIM	Required keyword.
<variable_name>	Name that begins with a letter (A-Z or a-z) and whose remaining characters can be any combination of letters (A-Z or a-z), digits (0-9) and underscores (_). The variable name is limited to 255 Unicode characters.
<upper_bound>	Reference to the upper bound (a number between 1 and 2,147,483,647, inclusive) of an array dimension. Three dimensions are supported in a DIM statement, each being nested in the syntax structure. After the upper bound is specified, it is fixed after the declaration. A statement similar to Visual Basic's ReDim is not supported.  The lower bound of each array dimension is always 1.



---

AS	Optional keyword for declaring the variable's datatype.
----	---

---

<data_type>	<p>Any one of the following 11 datatypes: Boolean, Discrete, Integer, ElapsedTime, Float, Real, Double, String, Message, Time or Object.</p> <p>Data_type can also be a .Net data_type like System.Xml.XmlDocument or a type defined in an imported script library</p> <p>If you omit the AS clause from the DIM statement, the variable, by default, is declared as an Integer datatype. For example:</p> <pre>DIM LocVar1;</pre> <p>is equivalent to:</p> <pre>DIM LocVar1 AS Integer;</pre>
-------------	--

---

In contrast to attribute names, variable names must not contain dots. Variable names and the data type identifiers are not case sensitive. If there is a naming conflict between a declared variable and another named entity in the script (for example, attribute name, alias or name of an object leveraged by the script), the variable name takes precedence over the other named entities. If the variable name is the same as an alias name, a warning message appears when the script is validated to indicate that the alias is ignored.

The syntax for specifying the entire array is “[ ]” for both local array variables and for attribute references. For example, to assign an attribute array to a local array, the syntax is:

```
locarr[] = tag.attr[];
```

DIM statements can be located anywhere in the script body, but they must precede the first referencing script statement or expression. If a local variable is referenced before the DIM statement, script validation done when you save the object containing the script prompts you to define it.

---

**Caution:** The validation mentioned above occurs only when you save the object containing the script. This is not the script syntax validation done when you click the **Validate Script** button.

---

Do not cascade DIM statements. For example, the following examples are invalid:

```
DIM LocVar1 AS Integer, LocVar2 AS Real;  
DIM LocVar3, LocVar4, LocVar5, AS Message;
```

To declare multiple variables, you must enter separate DIM statements for each variable.

When used on the right side of an equation, declared local variables always cause expressions on the left side to have Good quality. For example :

```
dim x as integer;  
dim y as integer;  
x = 5;  
y = 5;  
me.attr = 5;  
me.attr = x;  
me.attr = x+y;
```

In each case of `me.attr`, quality is Good.

When you use a variable in an expression to the right of the operator, its Quality is treated as Good for the purpose of data quality propagation.

You can use null to indicate that there is no object currently assigned to a variable. Using null has the same meaning as the keyword "null" in C# or "nothing" in Visual Basic. Assigning null to a variable makes the variable eligible for garbage collection. You may not use a variable whose value is null. If you do, the script terminates and an error message appears in the logger. You may, however, test a variable for null. For example:

```
IF myvar == null THEN ...
```

It is not possible to pass attributes as parameters for system objects. To work around this issue, use a local variable as an intermediary or explicitly convert the attribute to a string using an appropriate function call when calling the system object.

## Numbers and Strings

Allowed format for integer constants in decimal format is as follows:

```
IntegerConst = 0 or [sign] <non-zero_digit> <digit>*;
```

where:

```
sign ::= + | -
```

```
non-zero_digit ::= 1-9
```

```
digit ::= 0-9
```

For example, an integer constant is a zero or consists of an optional sign followed by one or more digits. Leading zeros are not allowed. Integer constants outside the range  $-2147483648$  to  $2147483647$  cause an overflow error.

Prepending either 0x or 0X causes a literal integer constant to be interpreted as hexadecimal notation. The +/- sign is supported.

The acceptable float for integers in hexadecimal is as follows:

```
IntegerHexConst = [<sign>] <0><x (or X)> <hexdigit>*
```

where:

```
sign ::= + or -
```

```
hexdigit ::= 0-9, A-F, a-f (only eight hexdigits [32-bits] are allowed)
```

Allowed format for floats is as follows:

```
FloatConst ::= [<sign>] <digit>* .<digit>+ [<exponent>;]
```

or

```
[<sign>] <digit>+ [.<digit>* [<exponent>]];
```

where:

```
sign ::= + or -
```

```
digit ::= 0-9 (can be one or more decimal digits)
```

```
exponent = e (or E) followed by a sign and then digit(s)
```

Float constants are applicable as values for variables of type float, real, or double. For example, float constants do not take the number of bytes into account. Script validation detects an overflow when a float, real, or double variable has been assigned a float constant that exceeds the maximum value.

If no digits appear before the period (.), at least one must appear after it. If neither an exponent part nor the period appears, a period is assumed to follow the last digit in the string.

If an attribute reference exists that has a format similar to a float constant with an exponent (such as “5E3”), then use the Attribute qualifier, as follows:

```
Attribute("5E3")
```

Strings must be surrounded by double quotation marks. They are referred to as quoted strings. The double-double quote indicates a single double-quote in the string. For example, the string:

```
Joe said, "Look at that."
```

can be represented in QuickScript .NET as:

```
"Joe said, ""Look at that."""
```

## QuickScript .NET Control Structures

QuickScript .NET provides five primary control structures in the scripting environment:

- IF ... THEN ... ELSEIF ... ELSE ... ENDIF
- FOR ... TO ... STEP ... NEXT Loop
- FOR EACH ... IN ... NEXT
- TRY ... CATCH
- WHILE Loop

### IF ... THEN ... ELSEIF ... ELSE ... ENDIF

IF-THEN-ELSE-ENDIF conditionally executes various instructions based on the state of an expression. The syntax is as follows:

```
IF <Boolean_expression> THEN;  
    [statements];  
[ { ELSEIF;  
    [statements] } ];  
[ ELSE;  
    [statements] ];  
ENDIF;
```

Where `Boolean_expression` is an expression that can be evaluated as a Boolean.

Depending on the data type returned by the expression, the expression is evaluated to constitute a True or False state according to the following table:

<b>Data Type</b>	<b>Mapping</b>
Boolean, Discrete	Directly used (no mapping needed).
Integer	Value = 0 evaluated as False Value != 0 evaluated as True.
Float, Real	Value = 0 evaluated as False Value != 0 evaluated as True.
Double	Value = 0 evaluated as False Value != 0 evaluated as True.
String, Message	Cannot be mapped. Using an expression that results in a string type as the Boolean_expression results in a script validation error.
Time	Cannot be mapped. Using an expression that results in a time type as the Boolean_expression results in a script validation error.
ElapsedTime	Cannot be mapped. Using an expression that results in an elapsed time type as the Boolean_expression results in a script validation error.
Object	Using an expression that results in an object type. Validates, but at run time, the object is converted to a Boolean. If the type cannot be converted to a Boolean, a run-time exception is raised.

The first block of statements is executed if Boolean\_expression evaluates to True. Optionally, a second block of statements can be defined after the keyword ELSE. This block is executed if the Boolean\_expression evaluates to False.

To help decide between multiple alternatives, an optional ELSEIF clause can be used as often as needed. The ELSEIF clause mimics switch statements offered by other programming languages.

Example:

```
IF value == 0 Then
    Message = "Value is zero";
ELSEIF value > 0 Then;
    Message = "Value is positive";
```

```
        ELSEIF value < 0 then;
            Message = "Value is negative";
        ELSE;
            {Default. Should never occur in this example};
    ENDIF;
```

The following syntax is also supported:

```
IF <Boolean_expression> THEN;
    [statements];
[ { ELSEIF;
    [statements] } ];
[ ELSE;
    [statements] ];
ENDIF;
ENDIF;
```

This approach nests a brand new IF compound statement within a previous one and requires an additional ENDIF.

See "Sample Scripts" on page 121 for more ideas about using this type of control structure.

## IF ... THEN ... ELSEIF ... ELSE ... ENDIF and Attribute Quality

When an attribute value is copied to another attribute of the same type, the attribute's quality is also copied. This can be especially relevant when working with I/O attributes. For example, the following two statements copy both value and quality:

```
me.Attr2 = me.Attr1;
me.Attr2.value = me.Attr1.value;
```

If only the value needs to be copied and the attribute has the quality BAD, you can use a temporary variable to hold the value. For example:

```
Dim temp as Integer;
temp = me.Attr1;
me.Attr2 = temp;
```

If there is a comparison such as `Attr1 <> Attr2` and one of the attributes has the BAD quality BAD, then the statements within the IF control block are not executed. For example, assuming `Attr1` has the quality BAD:

```
if me.Attr1<> me.Attr2 then
    me.Attr2 = me.Attr1;
endif;
```

In this script, the statement `me.Attr2 = me.Attr1` is not executed because `Attr1` has the quality BAD and comparing a BAD quality value with a good quality value is not defined/not possible.

The recommended approach is to first verify the quality of `Attr1`, as shown in the following example:

```
if(IsBad(me.Attr1)) then

LogMessage("Attr1 quality is bad, its value is not copied to
  Attr2");

else

  if me.Attr1<> me.Attr2 then

    me.AttrA2 = me.Attr1;

  endif;

endif;
```

An alternative method of verifying quality is to use the "==" operator:

```
if Me.Attr1 == TRUE then
```

Or, you can add the "value" property to the simplified IF THEN statement:

```
if Me.Attr1.value then
```

Using any of the above methods to verify data quality will ensure that your scripts execute correctly.

## FOR ... TO ... STEP ... NEXT Loop

FOR-NEXT performs a function (or set of functions) within a script several times during a single execution of a script. The general format of the FOR-NEXT loop is as follows:

```
FOR <analog_var> = <start_expression> TO <end_expression> [STEP
  <change_expression>];

  [statements];

  [EXIT FOR;];

  [statements];

NEXT;
```

Where:

- `analog_var` is a variable of type Integer, Float, Real, or Double.
- `start_expression` is a valid expression to initialize `analog_var` to a value for execution of the loop.
- `end_expression` is a valid expression. If `analog_var` is greater than `end_expression`, execution of the script jumps to the statement immediately following the NEXT statement.

This holds true if loop is incrementing up, otherwise, if loop is decrementing, loop termination occurs if `analog_var` is less than `end_expression`.

- `change_expression` is an expression that defines the increment or decrement value of `analog_var` after execution of the `NEXT` statement. The `change_expression` can be either positive or negative.
  - If `change_expression` is positive, `start_expression` must be less than or equal to `end_expression` or the statements in the loop do not execute.
  - If `change_expression` is negative, `start_expression` must be greater than or equal to `end_expression` for the body of the loop to be executed.
- If `STEP` is not set, then `change_expression` defaults to 1 for increasing increments, and defaults to -1 for decreasing increments.

Exit the loop from within the body of the loop with the `EXIT FOR` statement.

The `FOR` loop is executed as follows:

- 1 `analog_var` is set equal to `start_expression`.
- 2 If `change_expression` is positive, the system tests to see if `analog_var` is greater than `end_expression`. If so, the loop exits. If `change_expression` is negative, the system tests to see if `analog_var` is less than `end_expression`. If so, program execution exits the loop.
- 3 The statements in the body of the loop are executed. The loop can potentially be exited via the `EXIT FOR` statement.
- 4 `analog_var` is incremented by 1, -1, or by `change_expression` if it is specified.
- 5 Steps 2 through 4 are repeated.

---

**Note:** `FOR-NEXT` loops can be nested. The number of levels of nesting possible depends on memory and resource availability.

---

See "Sample Scripts" on page 121 for ideas about using this type of control structure.



## FOR EACH ... IN ... NEXT

FOR EACH loops can be used only with collections exposed by OLE Automation servers. A FOR-EACH loop performs a function (or set of functions) within a script several times during a single execution of a script. The general format of the FOR-EACH loop is as follows:

```
FOR EACH <object_variable> IN <collection_object >
    [statements];
    [EXIT FOR;];
    [statements];
NEXT;
```

Where:

- `object_variable` is a dimmed variable.
- `collection_object` is a variable holding a collection object.

As in the case of the FOR ... TO loop, it is possible to exit the execution of the loop through the statement EXIT FOR from within the loop.

See "Sample Scripts" on page 121 for ideas about using this type of control structure.

## TRY ... CATCH

TRY ... CATCH provides a way to handle some or all possible errors that may occur in a given block of code, while still running rather than terminating the program. The TRY part of the code is known as the try block. Deal with any exceptions in the CATCH part of the code, known as the catch block.

The general format for TRY ... CATCH is as follows:

```
TRY
    [try statements] 'guarded section
CATCH
    [catch statements]
ENDTRY
```

Where:

*tryStatements*

Statement(s) where an error can occur. Can be a compound statement. The *tryStatement* is a guarded section.

*catchStatements*

Statement(s) to handle errors occurring in the associated Try block. Can be a compound statement.

---

**Note:** Statements inside the Catch block may reference the reserved ERROR variable, which is a .NET System.Exception thrown from the Try block. The statements in the Catch block run only if an exception is thrown from the Try block.

---

TRY ... CATCH is executed as follows:

- 1 Run-time error handling starts with TRY. Put code that might result in an error in the try block.
- 2 If no run-time error occurs, the script will run as usual. Catch block statements will be ignored.
- 3 If a run-time error occurs, the rest of the try block does not execute.
- 4 When a run-time error occurs, the program immediately jumps to the CATCH statement and executes the catch block.

The simplest kind of exception handling is to stop the program, write out the exception message, and continue the program.

The error variable is not a string, but a .NET object of System.Exception. This means you can determine the type of exception, even with a simple CATCH statement. Call the GetType() method to determine the exception type, and then perform the operation you want, similar to executing multiple catch blocks.

**Example:**

```
dim command = new System.Data.SqlClient.SqlCommand;
dim reader as System.Data.SqlClient.SqlDataReader;
command.Connection = new System.Data.SqlClient.SqlConnection;

try
    command.Connection.ConnectionString = "Integrated
    Security=SSPI";
    command.CommandText="select * from sys.databases";
    command.Connection.Open();
    reader = command.ExecuteReader();

    while reader.Read()
        me.name = reader.GetString(0);
        LogMessage(me.name);
    endwhile;
catch
    LogMessage(error);
endtry;

if reader <> null and not reader.IsClosed then
    reader.Close();
endif;

if command.Connection.State == System.Data.ConnectionState.Open
then
    command.Connection.Close();
```

```
endif;
```

## WHILE Loop

WHILE loop performs a function or set of functions within a script several times during a single execution of a script while a condition is true. The general format of the WHILE loop is as follows:

```
WHILE <Boolean_expression>
    [statements]
    [EXIT WHILE;]
    [statements]
ENDWHILE;
```

Where: `Boolean_expression` is an expression that can be evaluated as a Boolean as defined in the description of IF...THEN statements.

It is possible to exit the loop from the body of the loop through the EXIT WHILE statement.

The WHILE loop is executed as follows:

- 1 The script evaluates whether the `Boolean_expression` is true or not. If not, program execution exits the loop and continues after the ENDWHILE statement.
- 2 The statements in the body of the loop are executed. The loop can be exited through the EXIT WHILE statement.
- 3 Steps 1 through 2 are repeated.

---

**Note:** WHILE loops can be nested. The number of levels of nesting possible depends on memory and resource availability.

---

See "Sample Scripts" on page 121 for ideas about using this type of control structure.

## QuickScript .NET Operators

The following QuickScript .NET operators require a single operand:

Operator	Short Description
~	Complement
-	Negation
NOT	Logical NOT

The following QuickScript .NET operators require two operands:

<b>Operator</b>	<b>Short Description</b>
+	Addition and concatenation
-	Subtraction
&	Bitwise AND
*	Multiplication
**	Power
/	Division
^	Exclusive OR
	Inclusive OR
<	Less than
<=	Less than or equal to
<>	Not equal to
=	Assignment
==	Equivalency (is equivalent to); not supported for entire array compares. Arrays must be compared one element at a time using ==.
>	Greater than
>=	Greater than or equal to
AND	Logical AND
MOD	Modulo
OR	Logical OR
SHL	Left shift
SHR	Right shift

Precedence of operators are shown below:

<b>Precedence</b>	<b>Operator</b>
1 (highest)	( )
2	- (negation), NOT, ~
3	**
4	*, /, MOD
5	+, - (subtraction)
6	SHL, SHR

Precedence	Operator
7	<, >, <=, >=
8	==, <>
9	&
10	^
11	
12	=
13	AND
14 (lowest)	OR

Arguments for the previously listed operators can be numbers or attribute values. Putting parentheses around the arguments is optional. The operator names are not case-sensitive.

## Parentheses ( )

Parentheses specify the correct order of evaluation for the operator(s). They can also make a complex expression easier to read. Operator(s) in parentheses are evaluated first, preempting the other rules of precedence that apply in the absence of parentheses. If the precedence is in question or needs to be overridden, use parentheses.

In the example below, parentheses add B and C together before multiplying by D:

```
( B + C ) * D;
```

## Negation ( - )

Negation is an operator that acts on a single component. It converts a positive integer or real number into a negative number.

## Complement ( ~ )

This operator yields the one's complement of a 32-bit integer. It converts each zero-bit to a one-bit and each one-bit to a zero-bit. The one's complement operator is an operator that acts on a single component, and it accepts an integer operand.

## Power ( \*\* )

The Power operator returns the result of a number (the base) raised to the power of a second number (the power). The base and the power can be any real or integer numbers, subject to the following restrictions:

- A zero base and a negative power are invalid.  
Example: "0 \*\* - 2" and "0 \*\* -2.5"
- A negative base and a fractional power are invalid.  
Example: "-2 \*\* 2.5" and "-2 \*\* -2.5"
- Invalid operands yield a zero result.

The result of the operation should not be so large or so small that it cannot be represented as a real number. Example:

```
1 ** 1 = 1.0
3 ** 2 = 9.0
10 ** 5 = 100,000.0
```

## Multiplication ( \* ), Division ( / ), Addition ( + ), Subtraction ( - )

These binary operators perform basic mathematical operations. The plus (+) can also concatenate String datatypes.

For example, in the data change script below, each time the value of “Number” changes, “Setpoint” changes as well:

```
Number=1;
Setpoint.Name = "Setpoint" + Text(Number, "#" );
```

Where: The result is "Setpoint1."

## Modulo (MOD)

MOD is a binary operator that divides an integer quantity to its left by an integer quantity to its right. The remainder of the quotient is the result of the MOD operation. Example:

```
97 MOD 8 yields 1
63 MOD 5 yields 3
```

## Shift Left (SHL), Shift Right (SHR)

SHL and SHR are binary operators that use only integer operands. The binary content of the 32-bit word referenced by the quantity to the left of the operator is shifted (right or left) by the number of bit positions specified in the quantity to the right of the operator.

Bits shifted out of the word are lost. Bit positions vacated by the shift are zero-filled. The shift is an unsigned shift.

## Bitwise AND ( & )

A bitwise binary operator compares 32-bit integer words with each other, bit for bit. Typically, this operator masks a set of bits. The operation in this example “masks out” (sets to zero) the upper 24 bits of the 32-bit word. For example:

```
result = name & 0xff;
```

## Exclusive OR ( ^ ) and Inclusive OR ( | )

The ORs are bitwise logical operators compare 32-bit integer words to each other, bit for bit. The Exclusive OR compare the status of bits in corresponding locations. If the corresponding bits are the same, a zero is the result. If the corresponding bits differ, a one is the result.

Example:

```
0 ^ 0 yields 0
```

```
0 ^ 1 yields 1
```

```
1 ^ 0 yields 1
```

```
1 ^ 1 yields 0
```

The Inclusive OR examines the corresponding bits for a one condition. If either bit is a one, the result is a one. Only when both corresponding bits are zeros is the result a zero. For example:

```
0 | 0 yields 0
```

```
0 | 1 yields 1
```

```
1 | 0 yields 1
```

```
1 | 1 yields 1
```

## Assignment ( = )

Assignment is a binary operator which accepts integer, real, or any type of operand. Each statement can contain only one assignment operator. Only one name can be on the left side of the assignment operator.

Read the equal sign (=) of the assignment operator as “is assigned to” or “is set to.”

---

**Note:** Do not confuse the equal sign with the equivalency sign (==) used in comparisons.

---

## Comparisons ( < , > , <= , >= , == , <> )

Comparisons in IF-THEN-ELSE statements execute various instructions based on the state of an expression.

## AND, OR, and NOT

These operators work only on discrete attributes. If these operators are used on integers or real numbers, they are converted as follows:

- Real to Discrete: If real is 0.0, discrete is 0, otherwise discrete is 1.
- Integer to Discrete: If integer is 0, discrete is 0, otherwise discrete is 1.

If the statement is "`Disc1 = Real1 AND Real2;`" and `Real1` is 23.7 and `Real2` is 0.0, `Disc1` has 0 assigned to it, since `Real1` is converted to 1 and `Real2` is converted to 0.

When assigning the floating-point result of a mathematical operation to an integer, Application Server rounds the value to the nearest integer instead of truncating it. This means that an operation like `IntAttr = 32/60` results in `IntAttr` having a value of 1, not 0. If truncation is needed, use the `Trunc()` function.



# Chapter 3

## Sample QuickScript .NET Scripts

This section includes sample scripts to help you to understand the QuickScript .NET scripting language.

---

**Important Note:** The sample scripts provided with a number of the Application Server scripting functions should work as written in most Windows operating system and installed software environments, but might not work with all possible hardware, operating system, and software combinations. We recommend that you modify the example scripts as necessary to fit your specific environment.

---

---

**Important Note:** Some sample scripts include references to public websites as examples. You may need to replace those URLs with a current and verified URLs.

---

### Sample Scripts

The sample scripts include:

- Accessing an Excel Spreadsheet Using an Imported Type Library
- Accessing an Excel Spreadsheet Using CreateObject
- Accessing an Office XP Excel Spreadsheet Using an Imported Type Library
- Calling a Web Service to Get the Temperature for a Specified Zip Code
- Calling a Web Service to Send an E-mail Message

- Creating a Look-up Table and Doing a Look-up on It
- Creating an XML Document and Saving it to Disk
- Executing a SQL Parameterized INSERT Command
- Filling a String Array and Using It
- Filling a Two-Dimensional Integer Array and Using It
- Formatting a Number Using a .NET Format 'Picture'
- Formatting a Time Using a .NET Format 'Picture'
- Getting the Directories Under the C Drive
- Loading an XML Document from Disk and Doing Look-ups on It
- Querying a SQL Server Database
- Reading a Performance Counter
- Reading a Text File from Disk
- Sharing a SQL Connection or Any Other .NET Object
- Using DDE to Access an Excel Spreadsheet
- Using Microsoft Exchange to Send an E-mail Message
- Using SMTP to Send an E-mail Message
- Using Screen-Scraping to Get the Temperature for a City
- Creating a Look-up Table and Doing a Look-up on It
- Dynamically Binding an Indirect Variable to a Reference

## Accessing an Excel Spreadsheet Using an Imported Type Library

```
dim app as Excel._Application;
dim wb as Excel._Workbook;
dim ws as Excel._WorkSheet;
app = new Excel.Application;
wb = app.Workbooks.Add();
ws = wb.ActiveSheet;
ws.get_Range("A1").Value = 1000;
ws.get_Range("A2").Value = 1000;
ws.get_Range("A3").Value = "=A1+A2";
```

```
LogMessage(ws.get_Range("A3").Value);  
wb.Close(false);
```

## Accessing an Excel Spreadsheet Using CreateObject

```
dim app as object;  
dim wb as object;  
dim ws as object;  
app = CreateObject("Excel.Application");  
wb = app.Workbooks.Add();  
ws = wb.ActiveSheet;  
ws.Range("A1").value = 20;  
ws.Range("A2").value = 30;  
ws.Range("A3").value = "=A1*A2";  
LogMessage(ws.Range("A3").Value);  
wb.Close(false);
```

## Accessing an Office XP Excel Spreadsheet Using an Imported Type Library

```
dim app as Excel.Application;  
dim ws as Excel.Worksheet;  
dim wb as Excel.Workbook;  
dim a1 as Excel.Range;  
dim a2 as Excel.Range;  
dim a3 as Excel.Range;  
app = new Excel._ExcelApplicationClass;  
wb = app.ActiveWorkbook;  
ws = app.ActiveSheet;  
a1 = ws.Range("A1");  
a2 = ws.Range("A2");  
a3 = ws.Range("A3");  
a1.Value = 1000;  
a2.Value = 2000;
```

```
a3.Value = "=A1*A2";  
LogMessage(a3.Value);  
wb.Close(true, "c:\temp.xls", false);
```

## Calling a Web Service to Get the Temperature for a Specified Zip Code

---

**Note:** This sample script includes a reference to a public website as an example. You may need to replace that URL with a current and verified URL.

---

```
' Requires input string uda me.zipcode and output float uda  
me.temperature.  
  
' First, generate a wrapper for the web service (.Net SDK must  
be installed).  
  
' To generate wrapper, run the following commands from the DOS  
prompt:  
  
' set path=%path%;C:\Program Files\Microsoft Visual Studio  
.NET\FrameworkSDK\Bin  
  
' wsdl http://www.vbws.com/services/weatherretriever.asmx  
  
' csc /target:library WeatherRetriever.cs  
  
' Next import the generated WeatherRetriever.dll library into  
your galaxy.  
  
' Now write your script:  
  
dim wr as WeatherRetriever;  
wr = new WeatherRetriever;  
me.temperature = wr.GetTemperature(me.zipcode);
```

## Calling a Web Service to Send an E-mail Message

---

**Note:** This sample script includes a reference to a public website as an example. You may need to replace that URL with a current and verified URL.

---

```
' First, generate a wrapper for the web service (.Net SDK must  
be installed).  
  
' To generate wrapper, run the following commands from the DOS  
prompt:
```

```
' set path=%path%;C:\Program Files\Microsoft Visual Studio
.NET\FrameworkSDK\Bin

' wsdl /namespace:SendMail http://www.xml-
webservices.net/services/messaging/smtp_mail/mailsender.asmx

' csc /target:library Message.cs

' Next import the generated Message.dll library into your
galaxy.

' Now write your script:

dim m as SendMail.Message;
m = new SendMail.Message;
m.SendSimpleMail
(
{to:      } "<type valid email address here>",
{from:    } "<type valid email address here>",
{subject: } "Reminder to self",
{body:    } "Pick up eggs and milk on your way home."
);
```

## Creating a Look-up Table and Doing a Look-up on It

```
dim zipcodes as System.Collections.Hashtable;
zipcodes = new System.Collections.Hashtable;
zipcodes["Irvine"] = 92618;
zipcodes["Mission Viejo"] = 92692;
LogMessage(zipcodes["Irvine"]);
```

## Creating an XML Document and Saving it to Disk

```
dim doc      as System.Xml.XmlDocument;
dim catalog  as System.Xml.XmlElement;
dim book     as System.Xml.XmlElement;
dim title    as System.Xml.XmlElement;
dim author   as System.Xml.XmlElement;
```

```
dim lastName as System.Xml.XmlElement;
dim firstName as System.Xml.XmlElement;
' create new XML document rooted in catalog
doc = new System.Xml.XmlDocument;
catalog = doc.CreateElement("catalog");
doc.AppendChild(catalog);
' add a book to the catalog
book = doc.CreateElement("book");
title = doc.CreateElement("title");
author = doc.CreateElement("author");
lastName = doc.CreateElement("lastName");
firstName = doc.CreateElement("firstName");
author.AppendChild(lastName);
author.AppendChild(firstName);
book.AppendChild(title);
book.AppendChild(author);
catalog.AppendChild(book);
book.SetAttribute("isbn", "0385503822");
title.InnerText = "The Summons";
lastName.InnerText = "Grisham";
firstName.InnerText = "John";
' add another book
book = doc.CreateElement("book");
title = doc.CreateElement("title");
author = doc.CreateElement("author");
lastName = doc.CreateElement("lastName");
firstName = doc.CreateElement("firstName");
author.AppendChild(lastName);
author.AppendChild(firstName);
book.AppendChild(title);
book.AppendChild(author);
catalog.AppendChild(book);
book.SetAttribute("isbn", "044023722X");
title.InnerText = "A Painted House";
```

```
lastName.InnerText = "Grisham";  
firstName.InnerText = "John";  
' save the XML document to disk  
doc.Save("c:\catalog.xml");
```

## Executing a SQL Parameterized INSERT Command

```
dim connection as System.Data.SqlClient.SqlConnection;  
dim command as System.Data.SqlClient.SqlCommand;  
dim regionId as System.Data.SqlClient.SqlParameter;  
dim regionDesc as System.Data.SqlClient.SqlParameter;  
dim commandText as string;  
  
connection = new  
    System.Data.SqlClient.SqlConnection("server=(local);uid=sa;da  
        tabase=northwind");  
  
connection.Open();  
  
commandText = "INSERT INTO Region (RegionID, RegionDescription)  
    VALUES (@id, @desc)";  
  
command = new System.Data.SqlClient.SqlCommand(commandText,  
    connection);  
  
regionId = command.Parameters.Add("@id",  
    System.Data.SqlDbType.Int, 4);  
  
regionDesc = command.Parameters.Add("@desc",  
    System.Data.SqlDbType.NChar, 50);  
  
command.Prepare();  
  
regionId.Value = 5;  
regionDesc.Value = "Europe";  
  
command.ExecuteNonQuery();  
  
regionId.Value = 6;  
regionDesc.Value = "South America";  
  
command.ExecuteNonQuery();  
  
connection.Close();
```

## Filling a String Array and Using It

```
dim numbers[3] as string;
dim s as string;
numbers[1] = "one";
numbers[2] = "two";
numbers[3] = "three";
LogMessage(numbers[3]);
for each s in numbers[]
LogMessage(s);
next;
```

## Filling a Two-Dimensional Integer Array and Using It

```
dim x[2,3] as integer;
dim i as integer;
x[1, 1] = 1;
x[1, 2] = 2;
x[1, 3] = 3;
x[2, 1] = 4;
x[2, 2] = 5;
x[2, 3] = 6;
LogMessage(x[2, 3]);
for each i in x[]
LogMessage(i);
next;
```

## Formatting a Number Using a .NET Format 'Picture'

```
dim i as integer;
i = 1234;
LogMessage("Total cost: " + i.ToString("$#,###,###.00"));
```



## Formatting a Time Using a .NET Format 'Picture'

```
dim t as time;
t = Now();
LogMessage("The current time is: " + t.ToString("hh:mm:ss") +
    ".");
```

## Getting the Directories Under the C Drive

```
dim dir as System.IO.DirectoryInfo;
for each dir in System.IO.DirectoryInfo("c:\").GetDirectories()
LogMessage(dir.FullName);
next;
```

## Loading an XML Document from Disk and Doing Look-ups on It

```
dim doc as System.Xml.XmlDocument;
dim node as System.Xml.XmlNode;
doc = new System.Xml.XmlDocument;
doc.Load("c:\catalog.xml");
' find the title of the book whose isbn is 044023722X
node =
    doc.SelectSingleNode("/catalog/book[@isbn='044023722X']/title");
LogMessage(node.InnerText);
' find all titles written by Grisham
for each node in
    doc.SelectNodes("/catalog/book[author/lastName='Grisham']/title")
    LogMessage(node.InnerText);
next;
```

## Querying a SQL Server Database

```
dim connection as System.Data.SqlClient.SqlConnection;
dim command as System.Data.SqlClient.SqlCommand;
dim reader as System.Data.SqlClient.SqlDataReader;
connection = new
    System.Data.SqlClient.SqlConnection("server=(local);uid=sa;da
        tabase=northwind");
connection.Open();
command = new System.Data.SqlClient.SqlCommand("select * from
    customers", connection);
reader = command.ExecuteReader();
while reader.Read()
    LogMessage(reader("CompanyName"));
endwhile;
reader.Close();
connection.Close();
```

## Reading a Performance Counter

```
' Requires output float UDA me.PercentProcessorTime.
' Declarations
dim counter as System.Diagnostics.PerformanceCounter;
' Startup
counter = new System.Diagnostics.PerformanceCounter;
counter.CategoryName = "Processor";
counter.CounterName = "% Processor Time";
counter.InstanceName = "0";
' Execute
me.PercentProcessorTime = counter.NextValue();
```

## Reading a Text File from Disk

```
dim sr as System.IO.StreamReader;
sr = System.IO.File.OpenText("c:\MyFile.txt");
while sr.Peek() > -1
    LogMessage(sr.ReadLine());
endwhile;
sr.Close();
```

## Sharing a SQL Connection or Any Other .NET Object

In UserDefined\_001 do this:

```
dim connection as System.Data.SqlClient.SqlConnection;
' Startup
connection = new
    System.Data.SqlClient.SqlConnection("server=(local);uid=sa;da
    tabase=northwind");
connection.Open();
System.AppDomain.CurrentDomain.SetData
    ("NorthwindConnection", connection);
' Shutdown
System.AppDomain.CurrentDomain.SetData("NorthwindConnection",
    Null);
connection.Close();
```

Then in UserDefined\_002, UserDefined\_003, and so on, do this:

```
dim connection as System.Data.SqlClient.SqlConnection;
connection = System.AppDomain.CurrentDomain.GetData
    ("NorthwindConnection");
if connection <> null then
    System.Threading.Monitor.Enter(connection);
' use the connection
System.Threading.Monitor.Exit(connection);
endif;
```

## Using DDE to Access an Excel Spreadsheet

```
WWPoke("excel", "sheet1", "r1c1", "Hello");  
WWRequest("excel", "sheet1", "r1c1", me.Greeting);  
' Note: use "" to embed double quotation marks in strings  
WWExecute("excel", "sheet1",  
" [SELECT("R1C1")] [FONT.PROPERTIES(,"Bold")]");
```

## Using Microsoft Exchange to Send an E-mail Message

The following compatibility notes apply to use of this script:

- The script will not work on Microsoft Office 2010 or later due to changes in MAPI handling.
- The script is supported only for 32-bit versions of Microsoft Office.
- If you are using Microsoft Office 2007, you must download and install Microsoft Collaboration Data Objects (CDO) 1.2.1. Additional information on CDO 1.2.1 is available at: <http://www.microsoft.com/en-us/download/details.aspx?id=3671>

```
dim session as object;  
dim msg as object;  
dim sProfileInfo as string;  
sProfileInfo = "<type valid Microsoft Exchange Server Name  
here>" + StringChar(10) + "<type valid Exchange Server user  
name here>";  
session = CreateObject("MAPI.Session");  
session.Logon(, , False,False , , True, sProfileInfo);  
msg = session.Outbox.Messages.Add();  
msg.Recipients.Add("<type valid email address here>");  
msg.Recipients.Resolve();  
msg.Subject = "Reminder to self";  
msg.Text = "Pick up eggs and milk on your way home.";  
msg.Send();  
session.Logoff();
```

## Using Screen-Scraping to Get the Temperature for a City

**Note:** This sample script includes a reference to a public website as an example. You may need to replace that URL with a current and verified URL.

```
' Screen-scraping involves downloading a web page,
' then using a regular expression to retrieve the desired data.
' Requires input string UDA me.CityState, e.g. "Los Angeles,CA"
' and output float UDA me.temperature.

dim request as System.Net.WebRequest;
dim reader as System.IO.StreamReader;
dim regex as System.Text.RegularExpressions.Regex;
dim match as System.Text.RegularExpressions.Match;
request = System.Net.WebRequest.Create
(
"http://www.srh.noaa.gov/data/forecasts/zipcity.php?inputstring
=" +
System.Web.HttpUtility.UrlEncode(me.CityState)
);
reader = new
System.IO.StreamReader(request.GetResponse().GetResponseStream());
regex = new
System.Text.RegularExpressions.Regex("<br><br>(.*?)&deg;F<br>");
match = regex.Match(reader.ReadToEnd());
me.temperature = match.Groups(1);
```

## Using SMTP to Send an E-mail Message

```
System.Web.Mail.SmtpMail.Send
(
{from:    } "<type valid email address here>",
{to:     } "<type valid email address here>",
```

```
{subject: } "Reminder to self",  
{body:   } "Pick up eggs and milk on your way home."  
  
);
```

## Writing a Text File to Disk

```
dim sw as System.IO.StreamWriter;  
sw = System.IO.File.CreateText("C:\MyFile.txt");  
sw.WriteLine("one");  
sw.WriteLine("two");  
sw.WriteLine("three");  
sw.Close();
```

## Dynamically Binding an Indirect Variable to a Reference

You can dynamically bind a variable of type Indirect to an arbitrary reference string and then use it for get/set purposes. For example:

```
' Assume reference obj1.Attr1 has value of 7  
dim x as indirect;  
dim s as string;  
s = "obj1.Attr1";  
x.BindTo(s);      ' where s is any expression that returns a  
                  string.  
                  ' The string should be an ArcestrA reference.  
obj2.Attr2 = x;   ' sets obj2.Attr2 to the reference x is bound  
to  
                  ' (obj1.Attr1 in this example, which has value of 7)  
x = 1234; ' sets obj1.Attr1 (in this example) to 1234  
IF WriteStatus(x) == MxStatusOk THEN  
    ' ... do something  
endif;
```

You cannot use `.BindTo` with an Indirect local variable to an attribute on another engine.

An unbound indirect returns no data.

If the Galaxy has Advanced Communication Management enabled, we do not recommend that you use references that are part of an ActiveOnDemand DIObject scan group in a script with an Indirect. The reference activation process is not in sync with the script execution, so using a function such as the IsUseable() function always returns false.

For example, the following scripting is NOT recommended.

In the declarations section:

```
Dim pPump as Indirect;
```

In Script Body section

```
pPump.BindTo("Pump_001.State"); 'Pump_001 is part of a DIObject  
scan group that has ActiveOnDemand enabled
```

```
IF IsUsable(pPump)
```

```
THEN Do this...' this will not execute
```

```
ELSE
```

```
Do that...
```

```
ENDIF;
```

```
pPump.BindTo("Pump_002.State"); 'Pump_002 is part of a DIObject  
scan group that has ActiveOnDemand enabled
```

```
IF IsUsable(pPump)
```

```
THEN Do this...' this will not execute
```

```
ELSE
```

```
Do that...
```

```
ENDIF;
```

In the script, only Pump\_002 is executing all of the time.

---

**Important:** If you have an existing application that uses the same Indirect variable with scripting more than one time for the items extended to device integration (DI) items or for DI items directly, and you enable Advanced Communication Management in the IDE, these scripts behave differently or do not execute as expected.

---





# Glossary

This glossary defines common terms in the Application Server and ArcestrA architecture.

<b>application</b>	A collection of objects within a Galaxy Repository that performs an automation task. Synonymous with Galaxy. You can have one or more applications within a Galaxy Repository.
<b>ApplicationEngine (AppEngine)</b>	A real-time engine that hosts and executes the run-time logic contained within AutomationObjects.
<b>ApplicationObject Toolkit</b>	A programmer's tool that creates new ApplicationObject templates, including configuration and run-time implementations.
<b>ApplicationObject</b>	Represents some element of your application. This can include things automation process components like <ul style="list-style-type: none"> <li>• thermocouple      • pump                      • motor</li> <li>• valve                      • reactor                      • tank</li> </ul> <p>or associated application components like</p> <ul style="list-style-type: none"> <li>• function block      • PID loop                      • Sequential Function Chart</li> <li>• Ladder Logic program      • batch phase                      • SPC data sheet).</li> </ul>
<b>Application Server</b>	The name of the product inside FactorySuite that forms a central application backbone. It includes the Galaxy Repository, one IDE, a WinPlatform, an AppEngine, and a basic library of ApplicationObjects.
<b>Area</b>	A logical grouping of AutomationObjects that represents an area or unit of a plant. An area groups related AutomationObjects for alarm, history, and security purposes. It is represented by an Area AutomationObject.

<b>assignment</b>	The designation of a host for an AutomationObject. For example, an AppEngine AutomationObject is assigned to a WinPlatform AutomationObject.
<b>attribute</b>	An externally accessible data item of an AutomationObject.
<b>attribute reference string</b>	An unambiguous text string that references an attribute of an AutomationObject.
<b>AutomationObject</b>	A type of object that represents permanent things in your plant, such as hardware, software, or engines, as objects with user-designated, unique names within the Galaxy. AutomationObjects provide a standard way to create, name, download, execute, and monitor the represented component.
<b>base template</b>	A root template at the top of a derivational hierarchy. Unlike other templates, a base template is not derived from another template but developed with the ApplicationObject Toolkit and imported into a Galaxy.
<b>block read group</b>	A DAGroup triggered by the user or another object. It reads a block of data from the external data source and indicates the completion status.
<b>block write group</b>	A DAGroup triggered by the user or another object after all the required data items are set. The block of data is then sent to the external data device. When the block write is complete, it indicates the completion status.
<b>change log</b>	The history log tracking the life cycle activities of ArchedstrA, such as object creation, check-in/check-out, deployment, save, rename, undeploy, undo checkout, override checkout and assignment.
<b>check-in</b>	IDE operation for persisting changes to an object to the Galaxy Repository and for making a configured object available for other users to check-out and use.
<b>check-out</b>	IDE operation for editing an object, making it unavailable for other users to check-out.
<b>checkpoint</b>	The act of saving on disk the configuration, state, and all associated data necessary to support automatic restart of a running AutomationObject. The restarted object has the same configuration, state, and associated data as the last checkpoint image on disk.
<b>contained name</b>	The name of an object as it exists within the context of its container. For example, a valve object with the TagName “Valve101” can be contained within a reactor and given the ContainedName “Inlet” within the context of its container. The ContainedName must be unique within the context of the containing object.

---

<b>containment</b>	The concept of placing one or more AutomationObjects within another AutomationObject. This results in a collection of AutomationObjects organized in a hierarchy that matches the application model and allows for better naming and manipulation. After placed in another AutomationObject, the contained object takes on a new name in addition to its unique tagname, known as the HierarchicalName. This name includes the container name and its name within the context of its container. For example, a level transmitter called TIC101 can be placed within a container object called Reactor1 and given the name Level within it, resulting in the HierarchicalName Reactor1.Level.
<b>DAGroup</b>	A data access group associated with DeviceIntegration objects. It defines how communications is achieved with external data sources. It can contain subscription, block read, and block write scan groups.
<b>Data Access Server (DAServer)</b>	The server executable that interfaces with DINetwork Objects and DIDevice Objects in the ArchestrA environment or with any third-party client, using various client protocols including OPC, DDE and SuiteLink.
<b>Data Access Server Toolkit (DAS Toolkit)</b>	A developer tool to build Data Access Servers (DAServers).
<b>DAServer Manager</b>	The Microsoft Management Console (MMC) snap-in supplied by the DAServer that provides the required user interface for activation, configuration, and diagnosis of the DAServer.
<b>deployment</b>	The operation of creating an AutomationObject on its target PC. Includes installing the necessary software, the object's configuration data, and starting the object up.
<b>derivation</b>	The creation of a new template based on an existing template.
<b>derived template</b>	Any template with a parent template.
<b>DeviceIntegration object (DIOjects)</b>	An AutomationObject that represents the communication with external devices. DIOjects run on an AppEngine, and include DINetwork Objects and DIDevice Objects.
<b>DIDevice Object</b>	A representation of an actual external device (for example, a PLC or RTU) that is associated with a DINetwork Object.
<b>DINetwork Object</b>	A representation of a physical connection to a DIDevice Object by means of the Data Access Server.
<b>event record</b>	The data that is transferred about the system and logged when a defined event changes state. For example, an analog crosses its high level limit or an acknowledgement is made.
<b>export</b>	The act of generating a Package file (.aaPKG file extension) from persisted data in the Galaxy Database. You can import the resulting .aaPKG file into another Galaxy through the IDE import mechanism.

<b>framework</b>	The ArchestrA infrastructure consisting of a common set of services, components, and interfaces for creating and deploying AutomationObjects that collect, store, visualize, control, track, report, and analyze plant floor processes and information.
<b>Galaxy database</b>	The relational database containing all persistent configuration information for all objects in a Galaxy.
<b>Galaxy Repository</b>	The software sub-system consisting of one or more Galaxy Databases.
<b>Galaxy</b>	Your entire application. The complete ArchestrA system consisting of a single logical name space and a collection of WinPlatforms, AppEngines and objects. One or more networked PCs that constitute an automation system. It defines the name space that all components and objects live in and defines the common set of system level policies that all components and objects comply with.
<b>hierarchical name</b>	<p>The fully qualified name of a contained object, including the container object's TagName. For example, a valve object with a contained name of "Inlet" within a reactor named "Reactor1" would have "Reactor1.Inlet" as the HierarchicalName.</p> <p>The valve object also has a unique TagName distinct from its HierarchicalName, such as Valve101.</p>
<b>historical storage system</b>	The time series data storage system, that compresses and stores high volumes of time series data for latter retrieval. Application Server leverages the Wonderware Historian as its historical storage system.
<b>host</b>	An AutomationObject to which other objects are assigned (for example, a WinPlatform is a host for an AppEngine).
<b>import</b>	Adding templates or instances to the Galaxy Database from an external file.
<b>instance</b>	A uniquely configured representation of an application component based on a template.
<b>instantiation</b>	The creation of a new object based on a corresponding template.
<b>Integrated Development Environment (IDE)</b>	Consists of various configuration editors to configure the total system, for general framework use and for the creation of your application.
<b>Log Viewer</b>	A Microsoft Management Console (MMC) snap-in that provides a user interface for viewing messages reported to the LogViewer.
<b>Message Exchange</b>	The object-to-object messaging system.
<b>object</b>	Any template or instance found in a Galaxy Database. A common characteristic of objects is they are stored as separate components in the Galaxy Repository.

---

<b>off-scan</b>	The state of an AutomationObject that indicates it is idle and not ready to execute its normal runtime processing.
<b>on-scan</b>	The state of an AutomationObject in which it is performing its normal runtime processing based on a configured schedule.
<b>Package Definition File (.aaPDF)</b>	The standard description file that contains the configuration data and implementation code for a base template. File extension is typically .aaPDF.
<b>Package File (.aaPKG)</b>	The result of the export function of the IDE. The description file that contains the configuration data and implementation code for a template and/or instance. File extension is typically .aaPKG.
<b>PLC</b>	Programmable logic controller.
<b>properties</b>	Data common to all attributes of objects, such as name, value, quality, and data type.
<b>reference</b>	A string that refers to an object or to data within one of its attributes.
<b>scan group</b>	A DAGroup that requires only the update interval defined, and the data is retrieved at the requested rate.
<b>System Management Console (SMC)</b>	The central runtime system administration/management user interface.
<b>TagName</b>	The unique name given to an instance. For example, an object might have the following TagName: V1101.
<b>template</b>	An object containing configuration information and, optionally, the code modules to create new objects, including derived templates and instances.
<b>toolset</b>	A named collection of templates listed together in the IDE Template ToolBox.
<b>UserDefined object</b>	An AutomationObject created from the \$UserDefined template. This template does not have any application-specific attributes or logic. Therefore, you must define these attributes and associated logic.
<b>WinPlatform object</b>	An object that represents a single computer in a Galaxy, consisting of a systemwide message exchange component, a set of basic services, the operating system, and the physical hardware. This object hosts all AppEngines.



# Index

## A

Abs(), script function 28  
 access  
     Excel spreadsheets using an imported type library, example script 122  
     Excel spreadsheets using CreateObject 123  
     Office XP Excel spreadsheets using imported type library 123  
 ActivateApp(), script function 28  
 addition ( + ) 118  
 Advanced Communication Management  
     closing a client application window containing scripts 18  
     description 18  
     minimizing a client application window containing scripts 19  
     opening a client application window containing scripts 18  
     restoring a client application window containing scripts 19  
 AND 120  
 ApplicationEngine, definition 137  
 ApplicationObject Toolkit, definition 137  
 ApplicationObject, definition 137  
 ArcCos(), script function 30  
 ArcSin(), script function 31

ArcTan(), script function 31  
 Area, definition 137  
 assignment ( = ) 119  
 assignment, definition 138  
 attribute reference string, definition 138  
 attribute, definition 138  
 AutomationObject, definition 138

## B

base template, definition 138  
 Bitwise AND ( & ) 119  
 block read group, definition 138  
 block write group, definition 138

## C

call web services, get the temperature for a specified zip code 124  
 call web services, send email 124  
 check-in, definition 138  
 check-out, definition 138  
 checkpoint, definition 138  
 comparisons ( , =, ==, ) 119  
 complement ( ~ ) 117  
 concatenating  
     entries 12  
     memory types 118

- message types 118
- containment, definition 139
- converting 117
  - positive integers 117
  - real numbers 117
- Cos(), script function 32
- create
  - look-up table and do look-up 125
  - XML documents and save to disk 125
- CreateObject, access Excel spreadsheets 123
- CreateObject(), script function 32

## D

- DAGroup, definition 139
- DAS Toolkit, definition 139
- DAServer, definition 139
- Data Access Server Toolkit, definition 139
- Data Access Server, definition 139
- Data Quality 110
- deployment
  - memory load and scripts 15
  - resource load and scripts 15
- deployment timeout period, scripts 15
- deployment, definition 139
- derivation, definition 139
- derived template, definition 139
- DeviceIntegration Object, definition 139
- DIObject, definition 139
- DINetwork Object, definition 139
- DIOObjects, definition 139
- division (/) 118
- DText(), script function 33
- dynamic
  - reference scripting 16
  - referencing 16

## E

- errors 16
  - dynamic reference scripting 16
- event record, definition 139
- example script
  - access Excel spreadsheets using an imported type library 122
  - access Excel spreadsheets using CreateObject 123
  - access Office XP Excel spreadsheets using imported type library 123

- call web services to get the temperature for a specified zip code 124
- call web services to send email 124
- create look-up tables and do look-up 125
- create XML documents and save to disk 125
- execute a SQL parameterized INSERT command 127
- fill string arrays and use them 128
- fill two-dimensional integer arrays and use 128
- format numbers using a .NET format 'picture' 128
- format time using a .NET format 'picture' 129
- get the directories under C
  - 129
- load XML documents from disk and do look-ups 129
- query SQL server databases 130
- read performance counters 130
- read text files from disk 131
- share a SQL connection or any other .NET object 131
- use DDE to access an Excel spreadsheet 132
- use Exchange to send an email 132
- use screen-scraping to get the temperature for a city 133
- write a text file to disk 125
- exclusive OR (^) 119
- execute a SQL parameterized INSERT command 127
- Execute method 16
  - dynamic referencing 16
- Exp(), script function 34
- export, definition 139
- expressions, syntax 12

## F

- fill
  - string array and use 128
  - two-dimensional integer array and use 128
- FOR
  - EACH IN NEXT loop 113
  - TO STEP NEXT loop 111
- format
  - numbers using a .NET format 'picture' 128
  - time using a .NET format 'picture' 129



**G**

GetCPQuality script function 34  
 GetCPTimeStamp script function 35

**I**

IF THEN ELSEIF ELSE ENDIF loop 108  
   Data Quality 110  
 inclusive OR ( | ) 119  
 Int(), script function 36  
 IsBad(), script function 37  
 IsGood(), script function 37  
 IsInitializing(), script function 38  
 IsUncertain(), script function 39  
 IsUsable(), script function 39

**L**

load XML documents from disk and do look-ups 129  
 Log Viewer, definition 140  
 Log(), script function 40  
 Log10(), script function 41  
 LogDataChangeEvent() script function 42  
 LogMessage(), script function 44  
 LogN(), script function 41  
 loops  
   FOR EACH IN NEXT 113  
   FOR TO STEP NEXT 111  
   IF THEN ELSEIF ENDIF 108  
   while 115

**M**

memory load during deployment 15  
 message types, concatenating 118  
 Modulo (MOD) 118  
 multiplication ( \* ) 118

**N**

negation ( - ) 117  
 NOT 120  
 Now(), script function 47  
 numbers and strings 107

**O**

off-scan, definition 141  
 OnScan scripts system resources 15  
 on-scan, definition 141  
 OR 120  
 order of evaluation, operators 117

**P**

package definition files, definition 141  
 package files, definition 141  
 parentheses ( ) 117  
 PDF, definition 141  
 Pi(), script function 47  
 PKG, definition 141  
 PLC, definition 141  
 positive integers  
   converting 117  
   raising to the power 117  
 power ( \*\* ) 117

**Q**

Quality, Data 110  
 query SQL server databases 130  
 QuickScript .NET  
   control structures 108  
   operator(s) 115  
   variables 103

**R**

raising to the power 117  
   positive integers 117  
   real numbers 117  
 read  
   performance counters 130  
   text files from disk 131  
 real numbers 117  
 required syntax  
   expressions 12  
   scripts 12  
 resources and deployment 15  
 Round(), script function 47

**S**

Scan Group, definition 141  
 script  
   editing styles and syntax 11  
 script function  
   Abs() 28  
   ActivateApp() 28  
   ArcCos() 30  
   ArcSin() 31  
   ArcTan() 31  
   Cos() 32  
   CreateObject() 32

- DText() 33
- Exp() 34
- GetCPQuality() 34
- GetCPTimeStamp 35
- Int() 36
- IsBad() 37
- IsGood() 37
- IsInitializing() 38
- IsUncertain() 39
- IsUsable() 39
- Log() 40
- Log10() 41
- LogDataChangeEvent() 42
- LogMessage() 44
- LogN() 41
- Now() 47
- Pi() 47
- Round() 47
- SendKeys() 48
- SetAttributeVT() 50
- SetBad() 51
- SetGood() 51
- SetInitializing() 52
- SetUncertain() 52
- Sgn() 53
- Sin() 75
- Sqrt() 75
- StringASCII() 76
- StringChar() 76
- StringFromIntg() 80
- StringFromReal() 80
- StringFromTime() 81
- StringInString() 83
- StringLeft() 85
- StringLen() 85
- StringLower() 86
- StringMid() 87
- StringReplace() 88
- StringRight() 89
- StringSpace() 90
- StringTest() 90
- StringToIntg() 92
- StringToReal() 92
- StringTrim() 93
- StringUpper() 94
- Tan() 95
- Text() 95
- Trunc() 96
- Types category 27
- WriteStatus() 97
- WWControl() 98
- WWExecute() 98
- WWPoke() 100
- WWRequest() 101
- scripts
  - deployment timeout period 15
  - dynamic references 16
  - syntax 12
- SendKeys(), script function 48
- SetAttributeVT() script function 50
- SetBad(), script function 51
- SetGood(), script function 51
- SetInitializing(), script function 52
- SetUncertain(), script function 52
- Sgn(), script function 53
- share a SQL connection or any other .NET object 131
- shift
  - left (SHL) 118
  - right (SHR) 118
- SHL 118
- SHR 118
- Sin(), script function 75
- SMC, definition 141
- Sqrt(), script function 75
- startup scripts
  - and system resources 15
- StringASCII(), script function 76
- StringChar(), script function 76
- StringFromIntg(), script function 80
- StringFromReal(), script function 80
- StringFromTime(), script function 81
- StringInString(), script function 83
- StringLeft(), script function 85
- StringLen(), script function 85
- StringLower(), script function 86
- StringMid(), script function 87
- StringReplace(), script function 88
- StringRight(), script function 89
- StringSpace(), script function 90
- StringTest(), script function 90
- StringToIntg(), script function 92
- StringToReal(), script function 92
- StringTrim(), script function 93
- StringUpper(), script function 94
- subtraction ( - ) 118

system resources, startup scripts 15

## T

TagName, definition 141

Tan(), script function 95

Template, definition 141

Text(), script function 95

timestamps 42, 50

Toolset, definition 141

Trunc(), script function 96

Types category, script function 27

## U

use SMTP to send an email 133

## V

Vista security restrictions 28

## W

while loop 115

winplatform object, definition 141

write a text file to disk 125

WriteStatus(), script function 97

WWControl(), script function 98

WWExecute(), script function 98

WWPoke(), script function 100

WWRequest(), script function 101

