## Contents

### Chapter 1  Common Scripting Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script Editing Styles and Syntax</td>
<td>7</td>
</tr>
<tr>
<td>Required Syntax for Expressions and Scripts</td>
<td>7</td>
</tr>
<tr>
<td>Simple Scripts</td>
<td>8</td>
</tr>
<tr>
<td>Script Execution Types</td>
<td>8</td>
</tr>
<tr>
<td>Startup Scripts</td>
<td>8</td>
</tr>
<tr>
<td>OnScan Scripts</td>
<td>8</td>
</tr>
<tr>
<td>Execute Scripts</td>
<td>8</td>
</tr>
<tr>
<td>OffScan Scripts</td>
<td>9</td>
</tr>
<tr>
<td>Shutdown Scripts</td>
<td>9</td>
</tr>
<tr>
<td>Deployment Scripts</td>
<td>9</td>
</tr>
<tr>
<td>Working with QuickScript Editor Features</td>
<td>10</td>
</tr>
<tr>
<td>Color Indicators for Script Elements</td>
<td>10</td>
</tr>
<tr>
<td>Autocomplete</td>
<td>10</td>
</tr>
<tr>
<td>Accepting Autocomplete Suggestions</td>
<td>14</td>
</tr>
<tr>
<td>Multi-level Undo and Redo</td>
<td>14</td>
</tr>
<tr>
<td>Dynamic Referencing Considerations</td>
<td>17</td>
</tr>
<tr>
<td>Run-Time Client Script Behavior</td>
<td>17</td>
</tr>
<tr>
<td>Opening a Client Application Window</td>
<td>17</td>
</tr>
<tr>
<td>Closing a Client Application Window</td>
<td>17</td>
</tr>
<tr>
<td>Minimizing a Client Application Window</td>
<td>17</td>
</tr>
<tr>
<td>Maximizing or Restoring a Client Application Window</td>
<td>17</td>
</tr>
<tr>
<td>Visual Indication of Script Errors</td>
<td>18</td>
</tr>
<tr>
<td>Line Numbers</td>
<td>18</td>
</tr>
<tr>
<td>Log Functions</td>
<td>18</td>
</tr>
</tbody>
</table>

### Chapter 2  QuickScript .NET Functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script Functions</td>
<td>19</td>
</tr>
<tr>
<td>Graphic Client Functions</td>
<td>19</td>
</tr>
<tr>
<td>GetCPQuality()</td>
<td>19</td>
</tr>
<tr>
<td>GetCPTimeStamp()</td>
<td>20</td>
</tr>
<tr>
<td>HideGraphic()</td>
<td>20</td>
</tr>
<tr>
<td>HideSelf()</td>
<td>21</td>
</tr>
<tr>
<td>Logoff()</td>
<td>21</td>
</tr>
<tr>
<td>ShowContent()</td>
<td>22</td>
</tr>
<tr>
<td>ShowGraphic()</td>
<td>25</td>
</tr>
<tr>
<td>ShowLoginDialog()</td>
<td>34</td>
</tr>
<tr>
<td>InTouch Functions</td>
<td>35</td>
</tr>
<tr>
<td>AddPermission() Function</td>
<td>35</td>
</tr>
<tr>
<td>AttemptInvisibleLogon() Function</td>
<td>36</td>
</tr>
<tr>
<td>ChangePassword() Function</td>
<td>36</td>
</tr>
<tr>
<td>EnableDisableKeys() Function</td>
<td>37</td>
</tr>
<tr>
<td>FileCopy()</td>
<td>38</td>
</tr>
<tr>
<td>FileDelete()</td>
<td>39</td>
</tr>
<tr>
<td>FileMove()</td>
<td>39</td>
</tr>
</tbody>
</table>
### Application Server Scripting Guide

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileReadFields() Function</td>
<td>40</td>
</tr>
<tr>
<td>FileReadMessage() Function</td>
<td>41</td>
</tr>
<tr>
<td>FileWriteFields() Function</td>
<td>41</td>
</tr>
<tr>
<td>FileWriteMessage() Function</td>
<td>42</td>
</tr>
<tr>
<td>GetAccountStatus() Function</td>
<td>43</td>
</tr>
<tr>
<td>GetNodeName() Function</td>
<td>43</td>
</tr>
<tr>
<td>InfoAppName() Function</td>
<td>44</td>
</tr>
<tr>
<td>InfoDisk() Function</td>
<td>44</td>
</tr>
<tr>
<td>InfoFile() Function</td>
<td>45</td>
</tr>
<tr>
<td>InfoInTouchAppDir() Function</td>
<td>46</td>
</tr>
<tr>
<td>InfoResources() Function</td>
<td>46</td>
</tr>
<tr>
<td>InTouchVersion() Function</td>
<td>47</td>
</tr>
<tr>
<td>InvisibleVerifyCredentials() Function</td>
<td>47</td>
</tr>
<tr>
<td>IsAssignedRole() Function</td>
<td>48</td>
</tr>
<tr>
<td>LaunchTagViewer() Function</td>
<td>48</td>
</tr>
<tr>
<td>LogonCurrentUser() Function</td>
<td>49</td>
</tr>
<tr>
<td>PlaySound() Function</td>
<td>49</td>
</tr>
<tr>
<td>PostLogonDialog() Function</td>
<td>50</td>
</tr>
<tr>
<td>PrintScreen() Function</td>
<td>50</td>
</tr>
<tr>
<td>QueryGroupMembership() Function</td>
<td>51</td>
</tr>
<tr>
<td>Starting a Windows Application</td>
<td>51</td>
</tr>
<tr>
<td>SwitchDisplayLanguage() Function</td>
<td>52</td>
</tr>
<tr>
<td>TseGetClientId() Function</td>
<td>52</td>
</tr>
<tr>
<td>TseGetClientNodeName() Function</td>
<td>52</td>
</tr>
<tr>
<td>TseQueryRunningOnClient() Function</td>
<td>53</td>
</tr>
<tr>
<td>TseQueryRunningOnConsole() Function</td>
<td>53</td>
</tr>
</tbody>
</table>

### Math Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs()</td>
<td>53</td>
</tr>
<tr>
<td>ArcCos()</td>
<td>53</td>
</tr>
<tr>
<td>ArcSin()</td>
<td>54</td>
</tr>
<tr>
<td>ArcTan()</td>
<td>54</td>
</tr>
<tr>
<td>Cos()</td>
<td>54</td>
</tr>
<tr>
<td>Exp()</td>
<td>55</td>
</tr>
<tr>
<td>Int()</td>
<td>55</td>
</tr>
<tr>
<td>Log()</td>
<td>56</td>
</tr>
<tr>
<td>Log10()</td>
<td>56</td>
</tr>
<tr>
<td>LogN()</td>
<td>56</td>
</tr>
<tr>
<td>Pi()</td>
<td>57</td>
</tr>
<tr>
<td>Round()</td>
<td>57</td>
</tr>
<tr>
<td>Sgn()</td>
<td>58</td>
</tr>
<tr>
<td>Sin()</td>
<td>58</td>
</tr>
<tr>
<td>Sqrt()</td>
<td>58</td>
</tr>
<tr>
<td>Tan()</td>
<td>59</td>
</tr>
<tr>
<td>Trunc()</td>
<td>59</td>
</tr>
</tbody>
</table>

### Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActivateApp()</td>
<td>60</td>
</tr>
<tr>
<td>DateTimeGMT()</td>
<td>60</td>
</tr>
<tr>
<td>lsBad()</td>
<td>61</td>
</tr>
<tr>
<td>lsGood()</td>
<td>61</td>
</tr>
<tr>
<td>lsInitializing()</td>
<td>62</td>
</tr>
<tr>
<td>IsUncertain()</td>
<td>62</td>
</tr>
<tr>
<td>lsUsable()</td>
<td>63</td>
</tr>
<tr>
<td>LogCustom()</td>
<td>63</td>
</tr>
<tr>
<td>LogDataChangeEvent()</td>
<td>64</td>
</tr>
<tr>
<td>LogError()</td>
<td>65</td>
</tr>
<tr>
<td>LogMessage()</td>
<td>65</td>
</tr>
<tr>
<td>LogTrace()</td>
<td>66</td>
</tr>
</tbody>
</table>
### QuickScript .NET Operators

- Parentheses ( ) .................................................................................................................. 107
- Negation ( - ) ....................................................................................................................... 109
- Complement ( ~ ) .................................................................................................................. 109
- Power ( ** ) ......................................................................................................................... 109
- Multiplication ( * ), Division ( / ), Addition ( + ), Subtraction ( - ) ..................................... 109
- Modulo (MOD) ...................................................................................................................... 110
- Shift Left (SHL), Shift Right (SHR) ..................................................................................... 110
- Bitwise AND ( & ) ................................................................................................................. 110
- Exclusive OR (^) and Inclusive OR ( | ) ................................................................................ 110
- Assignment ( = ) .................................................................................................................. 110
- Comparisons ( <, >, <=, >=, ==, <> ) .................................................................................. 111
- AND, OR, and NOT ............................................................................................................... 111

### Chapter 3  Sample QuickScript .NET Scripts

- Accessing an Excel Spreadsheet Using an Imported Type Library ....................................... 113
- Accessing an Excel Spreadsheet Using CreateObject ............................................................ 114
- Calling a Web Service to Get the Temperature for a Specified Zip Code ............................. 114
- Calling a Web Service to Send an E-mail Message ................................................................. 115
- Creating a Look-up Table and Doing a Look-up on It ............................................................. 115
- Creating an XML Document and Saving it to Disk ................................................................. 115
- Executing a SQL Parameterized INSERT Command .............................................................. 116
- Filling a String Array and Using It ......................................................................................... 116
- Filling a Two-Dimensional Integer Array and Using It .......................................................... 117
- Formatting a Number Using a .NET Format ‘Picture’ ............................................................. 117
- Formatting a Time Using a .NET Format ‘Picture’ .................................................................. 117
- Getting the Directories Under the C Drive ............................................................................. 117
- Loading an XML Document from Disk and Doing Look-ups on It ....................................... 117
- Querying a SQL Server Database .......................................................................................... 117
- Reading a Performance Counter ............................................................................................ 118
- Reading a Text File from Disk ............................................................................................... 118
- Sharing a SQL Connection or Any Other .NET Object .......................................................... 118
- Using DDE to Access an Excel Spreadsheet ........................................................................ 119
- Using Microsoft Exchange to Send an E-mail Message ......................................................... 119
- Using Screen-Scraping to Get the Temperature for a City .................................................... 119
- Using SMTP to Send an E-mail Message .............................................................................. 120
- Writing a Text File to Disk ..................................................................................................... 120
- Dynamically Binding an Indirect Variable to a Reference .................................................... 120

### Index

- 107
- 109
- 109
- 109
- 109
- 110
- 110
- 110
- 110
- 110
- 111
- 111
- 113
- 113
- 114
- 114
- 115
- 115
- 115
- 116
- 116
- 117
- 117
- 117
- 117
- 117
- 117
- 118
- 118
- 118
- 119
- 119
- 119
- 120
- 120
- 120
- 123
Chapter 1

Common Scripting Environment

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

In This Chapter

Script Editing Styles and Syntax ................................................................. 7
Required Syntax for Expressions and Scripts .............................................. 7
Simple Scripts ............................................................................................. 8
Script Execution Types.................................................................................. 8
Working with QuickScript Editor Features .................................................. 10

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 102.

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 indention. 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:

\[ a = \frac{(b - c)}{(2 + x)} \times xyz; \]

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 supported by InTouch OMI.

- **Startup Scripts**
- **OnScan Scripts**
- **Execute Scripts**
- **OffScan Scripts**
- **Shutdown Scripts**
- **Deployment Scripts**

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 and 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

**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.

<table>
<thead>
<tr>
<th>Element</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keywords</td>
<td>Blue</td>
</tr>
<tr>
<td>Syntax highlighted while typing.</td>
<td></td>
</tr>
<tr>
<td>Comments (both single line and multi-line)</td>
<td>Green</td>
</tr>
<tr>
<td>Syntax highlighted while typing.</td>
<td></td>
</tr>
<tr>
<td>Strings</td>
<td>Purple</td>
</tr>
<tr>
<td>Syntax highlighted while typing.</td>
<td></td>
</tr>
<tr>
<td>Function names, numeric constants, operators, semicolons, dim variables, alias variables, and so on</td>
<td>Black</td>
</tr>
<tr>
<td>See descriptions for Attribute names and Reserved words.</td>
<td></td>
</tr>
<tr>
<td>Attributes, InTouch Tags, Reference Strings</td>
<td>Maroon, bold face</td>
</tr>
<tr>
<td>Reserved words</td>
<td>Red, non-bold face</td>
</tr>
<tr>
<td>.NET type names</td>
<td>Teal, non-bold face</td>
</tr>
</tbody>
</table>

**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 Ctrl+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.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxBoolean attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxInteger attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxFloat attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxDouble attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxString attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxTime attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxElapsedTime attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxReference attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxStatus attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxDataTypeEnum attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxSecurityClassification attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxDataQuality attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxQualifiedEnum attribute</td>
</tr>
<tr>
<td><img src="image" alt="Icon" /></td>
<td>MxQualifiedStruct attribute</td>
</tr>
<tr>
<td>Icon</td>
<td>Represents</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td><img src="image" alt="MxInternationalizedString attribute icon" /></td>
<td>MxInternationalizedString attribute</td>
</tr>
<tr>
<td><img src="image" alt=".Net Method icon" /></td>
<td>.Net Method</td>
</tr>
<tr>
<td><img src="image" alt=".Net Property icon" /></td>
<td>.Net Property</td>
</tr>
<tr>
<td><img src="image" alt=".Net Field or Variable icon" /></td>
<td>.Net Field or Variable</td>
</tr>
<tr>
<td><img src="image" alt=".Net Namespace icon" /></td>
<td>.Net Namespace</td>
</tr>
<tr>
<td><img src="image" alt=".Net Struct icon" /></td>
<td>.Net Struct</td>
</tr>
<tr>
<td><img src="image" alt=".Net Class icon" /></td>
<td>.Net Class</td>
</tr>
<tr>
<td><img src="image" alt=".Net Interface icon" /></td>
<td>.Net Interface</td>
</tr>
<tr>
<td><img src="image" alt=".Net Enumeration icon" /></td>
<td>.Net Enumeration</td>
</tr>
<tr>
<td><img src="image" alt=".Net Enum Value icon" /></td>
<td>.Net Enum Value</td>
</tr>
<tr>
<td><img src="image" alt="QuickScript Keyword icon" /></td>
<td>QuickScript Keyword</td>
</tr>
</tbody>
</table>

- **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.**
- If the attribute cannot be exactly or unambiguously returned, this icon will appear.
- Partial name example: For me.alarm.a1, typing "me.alar" will show the blue ball icon for alarm.
- MxNone example: input/output extension attribute WriteValue.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Represents</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Rectangle icon" /></td>
<td>Rectangle</td>
</tr>
<tr>
<td><img src="image" alt="Rounded rectangle icon" /></td>
<td>Rounded rectangle</td>
</tr>
<tr>
<td><img src="image" alt="Line icon" /></td>
<td>Line</td>
</tr>
<tr>
<td>Icon</td>
<td>Represents</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>+</td>
<td>Horizontal or vertical line</td>
</tr>
<tr>
<td>T</td>
<td>Text</td>
</tr>
<tr>
<td>o</td>
<td>Ellipse</td>
</tr>
<tr>
<td>j</td>
<td>Curve</td>
</tr>
<tr>
<td>o</td>
<td>Closed curve</td>
</tr>
<tr>
<td>☐</td>
<td>Button</td>
</tr>
<tr>
<td>☐</td>
<td>Polygon</td>
</tr>
<tr>
<td>↝</td>
<td>Polyline</td>
</tr>
<tr>
<td>☐</td>
<td>Connect</td>
</tr>
<tr>
<td>☐</td>
<td>Image</td>
</tr>
<tr>
<td>☐</td>
<td>Group or embedded symbol</td>
</tr>
<tr>
<td>☐</td>
<td>Alarm control</td>
</tr>
<tr>
<td>☐</td>
<td>Edit box</td>
</tr>
<tr>
<td>☐</td>
<td>Arc</td>
</tr>
<tr>
<td>☐</td>
<td>Pie</td>
</tr>
<tr>
<td>☐</td>
<td>Chord</td>
</tr>
<tr>
<td>☐</td>
<td>Circle</td>
</tr>
</tbody>
</table>
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.

Dynamic Referencing Considerations

Dynamic reference scripting is one the biggest causes of deployment failures of StartUp and OnScan execution types.
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.

   ![Reference Example](image)

   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:

```plaintext
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";
' Set the Ref.Done UDA to True so that 'this script will not fire again
'unless the Object is deployed
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 ArchestrA® 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 ArchestrA objects opens in WindowViewer, the following script events occur:

- Register all ArchestrA and InTouch references used in embedded symbol scripts, if not registered already.
- Advise all ArchestrA 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 ArchestrA objects is closed, the following script events occur:

- Execute OnHide scripts of all embedded symbols within the window.
- Stop running client scripts.
- Unadvise all ArchestrA and InTouch references in the Window if there are no other open windows using the references.
- Unregister all ArchestrA 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 ArchestrA objects is minimized in WindowViewer, the following script events occur:

- Stop running client scripts associated with ArchestrA objects embedded in the window.
- Unadvise all ArchestrA 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 ArchestrA and InTouch script references in the window, if not advised already.
- Execute named scripts if their trigger conditions are met.
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 the message 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()` on page 63
- `LogError()` on page 65
- `LogMessage()` on page 65
- `LogTrace()` on page 66
- `LogWarning()` on page 66

**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.
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.

In This Chapter

Script Functions .......................................................... 19
QuickScript .NET Variables ................................................. 99
QuickScript .NET Control Structures .................................... 102
QuickScript .NET Operators ............................................... 107

Script Functions

This section describes the script functions included in the InTouch OMI development environment. The function documentation is organized into a set of folders that represents the same organization of the functions in the Script Function Browser.

- Graphic Client Functions on page 19
- InTouch Functions on page 35
- Math Functions on page 53
- Miscellaneous Functions on page 60
- String Functions on page 80
- System Functions on page 96
- WWDDE Functions on page 97

The Type folder contains a set of Microsoft .NET script functions, which are not documented. Refer to Microsoft .NET documentation for descriptions of the functions.

Graphic Client Functions

Use graphic client functions to hide and show symbols, open and close popup windows, log in and log off users, or to query custom properties contained in a symbol.

GetCPQuality()

Returns the Quality value of a custom property. This function is available within any ArchestrA graphic client script, but is not supported in InTouch OMI ViewApps (works with InTouch HMI only).

If used in an InTouch OMI ViewApp, GetCPQuality() always returns a value of 0 (bad quality).
### Syntax

```plaintext
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**

```plaintext
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 ArchestrA graphic client script.

**Syntax**

```plaintext
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**

```plaintext
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 ArchestrA graphic client script.

**Category**

Graphic Client

**Syntax**

```plaintext
HideGraphic(string identity);
```
Parameter

Identity
The unique name of the instance that shows the graphic.

Examples

```csharp
HideGraphic("i1");
```
Where "i1" is string Identity.

```csharp
HideGraphic("InTouch:Window1");
```
Where "InTouch:Window1" is the string identity.

See Also

ShowGraphic() on page 25, HideSelf() on page 21

HideSelf()

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

Category

Graphic Client

Syntax

```csharp
HideSelf();
```

Remarks

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

Example

```csharp
HideSelf();
```

See Also

ShowGraphic() on page 25, HideGraphic() on page 20

Logoff()

Action script that automatically logs off the current user from a ViewApp.

Action scripts are graphic animations that are triggered by a user action such as a mouse click.

Category

Miscellaneous

Syntax

```csharp
LogOff();
```

Parameter

None

Trigger

On Left-Click/Key/Touch Down

Additional Information

A log off button can be added that uses the Logoff() method to allow the user to log off from the ViewApp.
Example

Logoff() ;

See Also

ShowLoginDialog() on page 34

ShowContent()

Loads a graphic or a layout into an InTouch OMI pane. This function is available within any A
techstryA
graphic client script.

To load a graphic into a modal or modeless popup window, use ShowGraphic() on page 25. You can use ShowGraphic() for both InTouch OMI and InTouch HMI ViewApps. ShowContent() is for InTouch OMI

Category

Graphic Client

Syntax

Dim contentInfo as aaContent.ContentInfo;
contentInfo.Content = "<ContentName>";
ShowContent( contentInfo );

Parameter

ContentInfo

Data Type

aaContent.ContentInfo

Example

Show content in a pane

dim contentInfo as aaContent.ContentInfo;

Dim cpValues [2] as aaContent.PropertyOverrideValue;
cpValues[1] = new aaContent.PropertyOverrideValue("CP1", "20", true);
cpValues[2] = new aaContent.PropertyOverrideValue("CP2", "Pump.PV.TagName", false);

contentInfo.Content = "S1";
contentInfo.ContentType = "Overview";
contentInfo.OwningObject = "Enterprise";
contentInfo PaneName = "Panel";
contentInfo.ScreenName = "Wall";
contentInfo.PropertyOverrideValues = cpValues;

contentInfo.SearchScope = aaContent.SearchScope.PrimaryScreen;

ShowContent ( contentInfo );

aaContent.ContentInfo Properties

ContentInfo is a predefined structure that contains the data members described in the following
table.
String properties can be a concatenation of string and/or custom properties.

Note: See "Terms," below, for definitions of Content Type, Layout, Pane, Screen Profile, Primary Screen, and Source Screen.
<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
<th>Data Type</th>
<th>Required/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>A unique name that specifies the content to be loaded into the pane. Content can be a symbol, a layout, or external content.</td>
<td>String</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Content can be a symbol, a layout, or external content.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Symbol (graphic), layout, and external content names are listed in the Graphic Toolbox and in the Toolbox tab of the Layout and ViewApp editors. Relative names, for example, &quot;Me.S1,&quot; can also be used to designate content.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>However, if the content is a symbol and you are using object wizards that include Symbol Wizard custom property selections, and the symbol has an owning object, use the symbol's absolute name. This allows the correct symbol configuration to be shown for the instance. See Owning Object, below, for more information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Content name must be unique. Application Server does not check for duplicated names. If Content is duplicated, open content with the same name is closed, and the content with the duplicated name is opened in its place. PropertyOverrides and other ContentInfo parameters are updated with any new specified values.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>contentInfo.Content = &quot;UserDefinedObject_001.Symbol_001&quot;;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ScreenName</td>
<td>Specifies the screen that contains a pane in which to place the content.</td>
<td>String</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td><strong>Additional information</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ScreenNames are configured in the Screen Profile Editor. See Screen Profiles in the System Platform Help for additional information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>contentInfo.ScreenName = &quot;Wall&quot;;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PaneName</td>
<td>Specifies the pane in which to place the content.</td>
<td>String</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td><strong>Additional information</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>PaneNames are configured in the Layout Editor. See Layouts in the System Platform Help for additional information.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>contentInfo.PaneName = &quot;Pane1&quot;;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ContentType</td>
<td>Specifies the content type, for example, &quot;Overview,&quot; &quot;Navigation,&quot; or &quot;Faceplate.&quot;</td>
<td>String</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td><strong>Additional information</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ContentType is matched against the Content Type parameter that can be set for a pane in the Layout Editor. Content Type is used to override the actual type of the specified Content. If ContentType is not specified, Content is examined for its type of content. See Layouts in the System Platform Help for additional information about content types.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Example</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>contentInfo.ContentType = &quot;Overview&quot;;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SearchScope</td>
<td>When ScreenName has not been specified, SearchScope specifies which screen or screens will be searched for a pane that matches the specified PaneName or content type.</td>
<td>Enum</td>
<td>Optional</td>
</tr>
</tbody>
</table>
SearchScope is an enum with the following values:

- **AllScreens** searches all screens, starting with the source screen, then the primary screen, and then any remaining screens in alphabetical order.

- **SourceScreen** searches only the screen from which ShowContent was called.

- **PrimaryScreen** searches only the screen designated in the Screen Profile as the primary screen.

**Embedded Layout:** If SearchScope is not specified and ShowContent was called from an embedded layout, ShowContent searches locally for a matching pane within the embedded layout. When SearchScope or ScreenName is specified, ShowContent searches for a pane globally. It uses the SearchScope setting to determine which screens within the Screen Profile associated with the ViewApp will be searched.

```csharp
contentInfo.SearchScope = aaContent.SearchScope.PrimaryScreen;
```

**Property Overrides**

PropertyOverrides sets custom property overrides if a graphic has been specified by the Content property. Each override must include the custom property name and the override value.

<table>
<thead>
<tr>
<th>Property Overrides</th>
<th>PropertyOverrides sets custom property overrides if a graphic has been specified by the Content property. Each override must include the custom property name and the override value.</th>
<th>Property Override ValuePair[] array</th>
<th>Optional</th>
</tr>
</thead>
</table>

**Additional Information**

Property overrides are specified as a key-value pair, with the property name enclosed in quotes.

```csharp
Dim cpValues[2] as aaContent.PropertyOverrideValue;
cpValues[1] = new aaContent.PropertyOverrideValue("CP1", "20", true);
cpValues[2] = new aaContent.PropertyOverrideValue("CP2", "Pump.PV.TagName", false);
```

**OwningObject**

Sets the owning object of the content shown by the ShowContent() script function.

```csharp
contentInfo.OwningObject = "Enterprise";
```

**Terms**

**Content type:** specifies the type of content represented by a pane.

**Layout:** consists of one or more rectangular areas called panes that contain content shown in a ViewApp. A layout is associated with a screen.
Embedded Layout: In the context of ShowContent, an embedded layout is a layout that ShowContent places inside a pane of a containing layout.

Pane: rectangular area of a layout that can hold a single piece of content.

Primary screen: represents the main screen of a workstation that will show a running ViewApp

Screen Profile: defines the physical characteristics of one or more client workstation screens that will show a running ViewApp and how these screens are arranged with respect to each other.

Source screen: screen from which ShowContent was called.

See Also

ShowGraphic() on page 25

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 ArchestrA graphic client script.

Category

Graphic Client

Syntax

Show a graphic within a pop-up window

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

Call an InTouch window

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

Parameter

GraphicInfo

Data Type

aaGraphic.GraphicInfo

Examples

Show graphic within a pop-up window

```vbnet
ShowGraphic (graphicInfo);
```

Show an InTouch window

```vbnet
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

```java
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"

Use an absolute name to specify the symbol name and owning object if you are using an Object Wizard with Symbol Wizard custom property selections. This allows the correct symbol configuration to be shown for the instance. See [Owning Object](#), below, for more information.

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"; cpl + ".Symbol_001", where the value of cpl = "Pump_001"; or Obj1.Str1 + ".Symbol_001", where the value of Obj.Str1 = "Pump_001".

### Examples

#### Graphic Toolbox Reference

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

#### Absolute Reference

```java
graphicInfo.GraphicName = "OwningObjectName.SymbolName";
```

### 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.

**Note:** The `OwningObject` property sets references for the graphic, but is not associated with the `GraphicName` property if the symbol is part of an Object Wizard. Therefore, if you are scripting a symbol with an owning object, specify the owning object name as part of the `GraphicName` property, for example, `UserDefined_001.Pump_001`.

**Example**
```csharp
graphicInfo.OwningObject = "UserDefined_001";
```

*HasTitleBar*
Determines if the graphic is shown with a title bar.

**Data Type**
Boolean

**Default Value**
True

**Example**
```csharp
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**
```csharp
graphicInfo.WindowTitle = "Graphic01";
```

*WindowType*
Specifies whether window type is modal or modeless.

**Data Type**
Enum

**Default Value**
Modeless

**Valid Range**
0, 1

**Enumerations**
### WindowType

<table>
<thead>
<tr>
<th>WindowType</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modal</td>
<td>0</td>
</tr>
<tr>
<td>Modeless</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Examples

```csharp
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**

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

#### Resizable

Determines if the pop-up window is resizable.

**Data Type**

Boolean

**Default Value**

False

**Example**

```csharp
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**

<table>
<thead>
<tr>
<th>WindowLocation</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Center</td>
<td>0</td>
</tr>
<tr>
<td>Above</td>
<td>1</td>
</tr>
<tr>
<td>TopLeftCorner</td>
<td>2</td>
</tr>
<tr>
<td>Top</td>
<td>3</td>
</tr>
<tr>
<td>TopRightCorner</td>
<td>4</td>
</tr>
</tbody>
</table>
### WindowLocation

<table>
<thead>
<tr>
<th>WindowLocation</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeftOf</td>
<td>5</td>
</tr>
<tr>
<td>LeftSide</td>
<td>6</td>
</tr>
<tr>
<td>RightSide</td>
<td>7</td>
</tr>
<tr>
<td>RightOf</td>
<td>8</td>
</tr>
<tr>
<td>BottomLeftCorner</td>
<td>9</td>
</tr>
<tr>
<td>Bottom</td>
<td>10</td>
</tr>
<tr>
<td>BottomRightCorner</td>
<td>11</td>
</tr>
<tr>
<td>Below</td>
<td>12</td>
</tr>
</tbody>
</table>

**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 "Working with the Show/Hide Graphics Script Functions," in the *Creating and Managing ArchestrA Graphics User Guide*.

**Examples**

```csharp
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**

<table>
<thead>
<tr>
<th>WindowRelativePosition</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desktop</td>
<td>0</td>
</tr>
<tr>
<td>Window</td>
<td>1</td>
</tr>
<tr>
<td>ClientArea</td>
<td>2</td>
</tr>
<tr>
<td>ParentGraphic</td>
<td>3</td>
</tr>
<tr>
<td>ParentElement</td>
<td>4</td>
</tr>
<tr>
<td>Mouse</td>
<td>5</td>
</tr>
<tr>
<td>DesktopXY</td>
<td>6</td>
</tr>
</tbody>
</table>
### WindowRelativePosition

<table>
<thead>
<tr>
<th>WindowRelativePosition</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>WindowXY</td>
<td>7</td>
</tr>
<tr>
<td>ClientAreaXY</td>
<td>8</td>
</tr>
</tbody>
</table>

#### Examples

```plaintext
graphicInfo.WindowRelativePosition = aaGraphic.WindowRelativePosition.<WindowRelativePosition>;
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

<table>
<thead>
<tr>
<th>RelativeTo</th>
<th>Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphic</td>
<td>0</td>
</tr>
<tr>
<td>DeskTop</td>
<td>1</td>
</tr>
<tr>
<td>CustomizedWidthHeight</td>
<td>2</td>
</tr>
</tbody>
</table>

#### 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

```plaintext
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.

```
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.

```
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

```
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 "Working with the Show/Hide Graphics Script Functions" in the Creating and Managing ArchestrA Graphics User 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

HideSelf() on page 21

ShowLoginDialog()

Action script that shows a login dialog box with fields to enter a username and password. A typical login interface includes a login button that is selected by the user to show the Login dialog box with fields to enter a username and password.

Action scripts are graphic animations that are triggered by a user action such as a mouse click.

Category

Miscellaneous

Syntax

ShowLoginDialog();

Parameter

None

Trigger

On Left-Click/Key/Touch Down

Additional Information

A log off button can be added that uses the Logoff() method to allow the user to log off from the ViewApp.
Example
ShowLoginDialog();

See Also
Logoff() on page 21

InTouch Functions

The following InTouch functions can be used within the ArchestrA Symbol Editor script editor. In all functions that specify tag names as parameters, you can use InTouch tags from your InTouch application.

Note: Using the Convert to ArchestrA Symbol option in InTouch scripts, may wrongly append the term "InTouch:" to the script function name. To avoid errors, remove the term "InTouch:" from the script function name.

AddPermission() Function

Assigns a certain InTouch access level to a given user group on the local system or on the domain. When a user belonging to that group logs on to the InTouch HMI after the AddPermission() function is called, he or she receives the specified access level.

Category
security

Syntax
DiscreteTag=AddPermission( "Domain", "Group", AccessLevel);

Arguments

Domain
Name of the domain or local computer in which the group is located.

Group
Windows user group.

AccessLevel
InTouch access level that you want to associate with the given group.

Remarks
Valid for operating system security only. When this function is called, it checks for the presence of the specified group in the specified domain or workgroup. If successful, TRUE is returned, and the specified Access Level is associated with the group for subsequent user log ons. In all other cases, (that is, if an invalid value is specified for any of the arguments) FALSE is returned.

This function is typically configured to run on application startup. It does not affect users that are currently logged on. Only users that log on after AddPermission() is successfully called receive the access level associated with their group.

Examples
DiscreteTag=AddPermission( "corporate_hq", "InTouchAdmins", 9000);
DiscreteTag=AddPermission( "johns01", "InTouchUsers", 5000);

See Also
PostLogonDialog(), InvisibleVerifyCredentials(), IsAssignedRole(), AttemptInvisibleLogon(), QueryGroupMembership()
**AttemptInvisibleLogon() Function**

The AttemptInvisibleLogon() function can be used in a script to log on a user to InTouch using the supplied credentials. The user is not required to enter a password or user ID.

**Category**
security

**Syntax**

```plaintext
DiscreteTag=AttemptInvisibleLogon( "UserId", "Password", "Domain" );
```

**Arguments**

- **UserId**
  A valid user account name.

- **Password**
  Password of the user.

- **Domain**
  Name of the local computer, workgroup, or domain to which the user belongs. This argument applies only if the current security type is operating system-based.

**Return Value**

Returns TRUE if authentication is successful. Otherwise, it returns FALSE.

**Remarks**

An attempt is made to log on to the InTouch HMI using the supplied credentials.

- If the logon attempt succeeds, then TRUE is returned and the $OperatorDomain, $OperatorName, $AccessLevel, and $Operator system tags are updated accordingly.
- If the log on attempt fails, then FALSE is returned, and the currently logged on user (if any) continues to be the current user.

The `Domain` argument is only valid for operating system-based security. If ArchestrA security mode is in use and if ArchestrA security is in turn using operating system-based security, the `UserId` argument should contain the fully qualified user name with domain name or computer name.

**Examples**

When security is operating system-based:

```plaintext
DiscreteTag=AttemptInvisibleLogon("UserId", "Password", "Domain");
```

When security is either InTouch-based or ArchestrA-based:

```plaintext
DiscreteTag=AttemptInvisibleLogon("UserId", "Password", "" );
```

**See Also**

PostLogonDialog(), InvisibleVerifyCredentials(), IsAssignedRole(), QueryGroupMembership(), AddPermission()

**ChangePassword() Function**

Shows the Change Password dialog box, allowing the logged on operator to change his/her password.

**Category**
security

**Syntax**

```plaintext
[Result=]ChangePassword();
```
**Return Value**
Returns one of the following integer values:

- 0 = Cancel was pressed.
- 1 = OK was pressed.

**Remarks**
If the operator uses a touch screen, the operator can use the alphanumeric keyboard to enter the new password.

**Example**
The following script can be placed on a button or called from a condition script or data change script.

```
Errmsg=ChangePassword();
```

### EnableDisableKeys() Function
Enables/disables key filters for the Alt, Escape, and Windows keys.

**Category**
View

**Syntax**
```
EnableDisableKeys(AltKey, EscKey, WinKey);
```

**Parameters**
- **AltKey**
  Integer to enable or disable key filters for the Alt key:
  - 1 = enable filter (disable Alt key)
  - 0 = disable filter (enable Alt key)
- **EscKey**
  Integer to enable or disable key filters for the Escape key:
  - 1 = enable filter (disable Esc key)
  - 0 = disable filter (enable Esc key)
- **WinKey**
  Integer to enable or disable key filters for the Windows key:
  - 1 = enable filter (disable Win key)
  - 0 = disable filter (enable Win key)

**Remarks**
Disabling the Alt key also disables the Win+L key combination (for locking the Windows desktop). Win+L is the shortcut for another combination of keys that involves the Alt key. Thus, disabling the Alt key also disables the shortcut for locking the Windows desktop.

Disabling the Esc key disables it for all actions.

**Example(s)**
```
EnableDisableKeys(0,0,0); // enable all three keys
EnableDisableKeys(1,1,1); // disable all three keys
EnableDisableKeys(0,0,1); // enable Alt and Escape keys, disable Windows key.
```
**FileCopy() Function**

Copies a source file to a destination file and returns a status result. This function may take a longer time to execute and is executed in multiple stages:

1. FileCopy() function is called and an immediate result is returned, indicating success or failure of the file copy initialization.
2. FileCopy() function executes the copy procedure in the background, and InTouch scripting continues execution while the file copying is in progress. You can monitor the file copying progress with an integer tag.
3. FileCopy() function returns a file copy result, indicating success or failure of the file copy procedure.

If the destination folder is not available (i.e., another computer on the network), the function waits for up to 10 seconds to time out, and then posts a message in the Logger.

**Note:** Do not use the FileCopy() function in asynchronous QuickFunctions.

**Syntax**

```plaintext
result = FileCopy (sourcefile, destfile, progressstag)
```

**Parameters**

- **sourcefile**
  Full path and file name of the file to be copied. A literal string value, message tagname, or string expression. You can use the wildcard characters ("*" and ?) in this parameter to copy just files matching a specified criteria. The path name can also be a UNC path name.

- **destfile**
  Full path and file name (or just path name) of the destination. A literal string value, message tagname, or string expression. The path name can also be a UNC path.

- **progressstag**
  Name of an integer tag enclosed in double quotes that will contain a value indicating the file copy progress. A literal string value, message tagname (such as a message tag containing the value "IntTag.Name") or string expression. The values have following meaning:
  - 0 - FileCopy() procedure is still in progress.
  - 1 - FileCopy() procedure has completed successfully.
  - -1 - FileCopy() procedure completed with errors.

**Return Value**

A value of -1, 0, or 1 indicating the following:
- 1 - FileCopy() function successfully called.
- 0 - Error when calling the FileCopy() function because another FileCopy() procedure is already in progress.
- -1 - Error when calling the FileCopy() function because of a non-existent source file or the destination is read only.

**Example(s)**

This script copies the file c:\MyData\output.log to the directory d:\archive and renames the file to output.txt. The progress of the file copy is written to the integer tag Monitor.

```plaintext
Status=FileCopy("c:\MyData\output.log", "d:\archive\output.txt", "Monitor");
```

This script copies all files with file ending .txt in the c:\root directory to the destination directory c:\Backup.

```plaintext
Status=FileCopy("c:\*.txt", "c:\Backup", "Monitor");
```

This script copies a file whose full path and file name is contained in the message tag LogFile to the destination directory c:\results\ and renames it to logxxx.txt where xxx is a timestamp.
FileDelete() Function

Deletes an individual file.

Syntax

result = FileDelete(filename)

Parameters

filename

The path name and file name of the file to delete. A literal string value, message tagname, or string expression. UNC path names are supported.

Remarks

Do not use the wildcard characters (*) and ?) with the FileDelete() function and do not use the FileDelete() function in asynchronous QuickFunctions.

The FileDelete() function does not delete directories.

Return Value

A value indicating success or failure of the file deletion:

1 - file is deleted successfully
0 - file is not deleted successfully. Possible causes are attempts to delete a read only or a non-existent file.

Example(s)

This script deletes the file c:\Data.txt and returns 1 if the file was found and deleted successfully.  
Status=FileDelete("c:\Data.txt");

FileMove() Function

Moves a source file to a destination file and returns a status result. It can be also used to rename a file. This function may take a longer time to execute and executes in multiple stages:

1. FileMove() function is called and an immediate result is returned, indicating success or failure of the file move initialization.
2. FileMove() function executes the move procedure in the background, InTouch scripting continues execution while the file moving is in progress. You can monitor the file moving progress with an integer tag.
3. FileMove() function returns a file move result, indicating success or failure of the file moving procedure.

Do not use the FileMove() function in asynchronous QuickFunctions.

Syntax

result = FileMove(sourcefile, destfile, progressstag)

Parameters

sourcefile

Full path and file name of the file to be moved. A literal string value, message tagname, or string expression. You can use the wildcard characters (*) and ?) in this parameter to move just files matching a specified criteria. The path name can also be a UNC path name.

destfile

Full path and file name (or just path name) of the destination. A literal string value, message tagname, or string expression. The path name can also be a UNC path.
ProgressTag
Name of an integer tag enclosed in double quotes that will contain a value indicating the file moving progress. A literal string value, message tagname (such as a message tag containing the value "IntTag") or string expression. The values have following meaning:

0 - FileMove() procedure is still in progress
1 - FileMove() procedure has completed successfully
-1 - FileMove() procedure completed with errors

Return Value
A value of -1, 0, or 1 indicating the following:
1 - FileMove() function successfully called
0 - Error when calling the FileMove() function because another FileMove() procedure is already in progress
-1 - Error when calling the FileMove() function. Possible errors are attempts to move a non-existent file.

Example(s)
This script moves the file c:\MyData\output.log to the directory d:\archive and renames the file to output.txt. The progress of the file moving is written to the integer tag Monitor.
```
Status=FileMove("c:\MyData\output.log","d:\archive\output.txt","Monitor");
```
This script moves all files with file ending .txt in the c:\ root directory to the destination directory c:\Backup.
```
Status=FileMove("c:\*.txt", "c:\Backup", "Monitor");
```
This script moves a file whose full path and file name is contained in the message tag LogFile to the destination directory c:\results and renames it to logxxx.txt where xxx is a timestamp.
```
Status=FileMove(LogFile, "c:\results\log" + $DateString + $TimeString + ".txt", "Monitor");
```

FileReadFields() Function
Reads the values contained in a csv file into a series of tagnames. You can use this function to load a set of tagname values.
Commas are the only supported delimiter.
This function can only be used for synchronous calls.

Syntax
```
[result = ] FileReadFields (filename, offset, starttag, numberoffields)
```

Parameters

filename
Name of the csv file to read the data from. A literal string value, a message tagname or a string expression.

offset
Location (in bytes) in the file to start reading. A literal integer value, integer tagname, or integer expression.

starttag
Name of the first tagname that receives the first read data item. The tagname must be enclosed with double quotes and end in a number, such as "MyTag1". A literal string value, message tagname (such as a message tagname containing the value "MyTag1"), or a string expression.
**FileReadFields() Function**

Number of data items to read from the csv file. A literal integer value, integer tagname, or integer expression. The first data item is read into the tagname defined in the starttag parameter, subsequent data items into tagnames with the incremented numeral suffix of the starttag parameter (MyTag1, MyTag2, MyTag3, ...).

**Return Value**

Optional new file offset (in byte) after reading the data. This can be used to read the next set of data.

**Example(s)**

This script reads the values "Flour" to RecipeTag1, 27.23 to RecipeTag2, 14 to RecipeTag3, and 1 to RecipeTag4, and returns the new file offset—if the csv file c:\set.csv contains the following data: Flour, 27.23, 14, 1 and if the following tags are defined: RecipeTag1:message, RecipeTag2:real, Recipe3:integer, RecipeTag4:discrete.

```
FileReadFields("c:\set.csv",0,"RecipeTag1",4);
```

**FileReadMessage() Function**

Reads a specified number of bytes (or one line) of string data from a file.

**Syntax**

```
[result = ] FileReadMessage (filename, offset, messagetag, charstoread)
```

**Parameters**

- **filename**
  Name of the file to read the data from. A literal string value, a message tagname, or a string expression.

- **offset**
  Location (in bytes) in the file to start reading from. A literal integer value, integer tagname, or integer expression.

- **messagetag**
  Message tagname that receives the first line or number of bytes from the file. Enclose the tagname with double quotes when using the function within the ArchestrA Symbol Editor Script Editor.

- **charstoread**
  Number of bytes to read from the file. Set it to 0 to read until the next line feed (LF) character. A literal integer value, integer tagname, or integer expression.

**Return Value**

Contains the new byte position after the read. You can use this for subsequent reads from the file.

**Example(s)**

This script reads the first line of data in the file c:\Data\File.txt to the message tagname MsgTag.

```
FileReadMessage ("c:\Data\File.txt",0,MsgTag, 0);
FileReadMessage ("c:\Data\File.txt",0,"InTouch:MsgTag", 0);
```

**FileWriteFields() Function**

Writes the values contained in a series of tagnames to a csv file. You can use this function to save a set of tagname values.

Commas are the only supported delimiter.

**Syntax**

```
[result = ] FileWriteFields (filename, offset, starttag, numberoffields)
```
Parameters

*filename*
Name of the csv file to write the data to. A new file is created if it does not previously exist. A literal string value, a message tagname, or a string expression.

*offset*
Location (in bytes) in the file to start writing to. Use -1 to write to the end of the file (append). A literal integer value, integer tagname, or integer expression.

*starttag*
Name of the first tagname that contains the first data item to be written. The tagname must be enclosed with double quotes and end in a number, such as "MyTag1". A literal string value, message tagname (such as a message tagname containing the value "MyTag1") or a string expression.

*numberoffields*
Number of data items to write to the csv file. A literal integer value, integer tagname, or integer expression. The first data item is written from the tagname defined in the starttag parameter to the file, subsequent data items from tagnames with the incremented numeral suffix of the starttag parameter (MyTag1, MyTag2, MyTag3, ...).

Return Value
Optional new file offset (in byte) after writing the data. This can be used to write the next set of data.

Example(s)
A series of InTouch tags is defined as follows:

<table>
<thead>
<tr>
<th>Tagname</th>
<th>Data Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RecipeTag1</td>
<td>Message</td>
<td>Flour</td>
</tr>
<tr>
<td>RecipeTag2</td>
<td>Real</td>
<td>27.23</td>
</tr>
<tr>
<td>RecipeTag3</td>
<td>Integer</td>
<td>14</td>
</tr>
<tr>
<td>RecipeTag4</td>
<td>Discrete</td>
<td>1</td>
</tr>
</tbody>
</table>

This script writes the values contained in RecipeTag1 to RecipeTag4 to the csv file c:\set.csv.

FileWriteFields("c:\set.csv",0,"RecipeTag1",4);

So that the file c:\set.csv will contain the following data:

Flour,27.23,14,1

FileWriteMessage() Function

Writes a specified number of bytes (or one line) of string data to a file.

Syntax

\([\text{result} = ]\) FileWriteMessage (filename, offset, messagetag, linefeed)

Parameters

*filename*
Name of the file to write the data to. A literal string value, a message tagname, or a string expression.

*offset*
Location (in bytes) in the file to start writing to. Set it to -1 to write data to the end of the file (append). A literal integer value, integer tagname, or integer expression.

*messagetag*
Message tagname that contains the data to be written to the file.
linefeed
Specify whether to write a line feed (LF) character after writing the data to the file. Set to 1 to write a line feed character; otherwise, set it to 0. A literal Boolean value, discrete tagname, or Boolean expression.

Return Value
Contains the new byte position after the write. You can use this for subsequent writes to the file.

Example(s)
This script writes the value of a message tagname MsgTag to the end of the file c:\Data\File.txt.
FileWriteMessage("c:\Data\File.txt",-1,MsgTag,1);

GetAccountStatus() Function
Returns the number of days until the user's password expires.

Category
security

Syntax
Result=GetAccountStatus(Domain, UserID);

Arguments

Domain
Name of the domain or local computer in which the user account is located.

UserID
Windows user account name that is part of the local computer, workgroup, or domain.

Return Value
This function also returns the following values:

<table>
<thead>
<tr>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Account password expired</td>
</tr>
<tr>
<td>-2</td>
<td>Account password never expires</td>
</tr>
<tr>
<td>-3</td>
<td>Account locked out</td>
</tr>
<tr>
<td>-4</td>
<td>Account disabled</td>
</tr>
<tr>
<td>-5</td>
<td>Account info failed</td>
</tr>
</tbody>
</table>

Remarks
Use this script function with operating system-based security. Do not use this function with the ArchestrA security mode.

If the GetAccountStatus() function is used with ArchestrA security, the script attempts to retrieve the account information directly from the domain controller. This works as long as the ArchestrA Galaxy Repository is using operating system security with the same domain.

Example(s)
Status = GetAccountStatus("Corporate_HQ","Operator");

GetNodeName() Function
Returns the node name of the computer.
GetNodeName (messagetag, nodenum);

Parameters

messagetag
Message tagname that will contain the node name. Enclose the tagname with double quotes when using the function within the ArchestrA Symbol Editor Script Editor.

nodenum
Number of characters to retrieve from the node name. A literal integer value, integer tagname, or integer expression in the range of 0 to 131.

Example(s)

This script retrieves the node name and assigns it to the NodeName message tagname.
GetNodeName(NodeName, 131);
GetNodeName("InTouch:NodeName", 131);

InfoAppTitle() Function

Returns the application title or Windows task list name of a specified application that is running.

Syntax

result = InfoAppTitle(appname)

Parameters

appname
Name of the application without the .exe extension. A literal string value, message tagname, or string expression.

Example(s)

This script returns "Calculator"
InfoAppTitle("calc")

This script returns "Microsoft Excel"
InfoAppTitle("excel")

InfoDisk() Function

Returns either the total or free space on a local or network disk drive.

Syntax

result = InfoDisk(drive, infotype, trigger);

Parameters

drive
The drive letter for which you want to retrieve information. Only the first character of a string is used. A literal string value, message tagname, string expression.

infotype
Specifies the information type. A literal integer value, integer tagname, or integer expression with following possible values:

1 - function returns total size of disk drive (in bytes)
2 - function returns free space of disk drive (in bytes)
3 - function returns total size of disk drive (in kilobytes)
4 - function returns free space of disk drive (in kilobytes)
trigger
A tagname (or expression) that acts as a trigger to recalculate the disk information. If the trigger value changes the disk information is recalculated. A discrete or analog taname, or a discrete or analog expression.

Remarks
The trigger tag only has meaning when the InfoDisk() function is used in an animation display link. If this function is used in a script, you can specify any literal numeric value, analog tagname, or numeric expression.

Example(s)
Use this script in an animation display link to show the free space of disk drive C and update the information every minute.

\[ \text{InfoDisk("C", 4, $Minute)} \]

InfoFile() Function

Returns various information on a file or directory.

Syntax
\[ \text{result = InfoFile (filename, infotype, trigger)} \]

Parameters

- **filename**
The full file name or directory name you want to retrieve information about. A literal string value, message tagname, or string expression. Can also include wildcard characters, such as "*" and "?".

- **infotype**
The type of information you want to retrieve about the specified file or directory. A literal integer value, integer tagname, or integer expression with following values and meaning:

  1 - Existence. The InfoFile() function returns 1 if the file exists, 2 if the file is a directory and 0 if the file or directory does not exist.

  2 - Size. The InfoFile() function returns the file size in bytes.

  3 - Creation timestamp. The InfoFile() function returns the time stamp as seconds that have passed since midnight January 1st 1970. Use the StringFromTimeLocal() function to convert this value to a message timestamp.

  4 - Wildcard Search Match. The InfoFile() function returns the number of files that match a specified wildcard search.

- **trigger**
A tagname (or expression) that acts as a trigger to recalculate the file information. If the trigger value changes, the file information is recalculated. A discrete or analog taname, or a discrete or analog expression.

Remarks
The trigger tag only has meaning when the InfoFile() function is used in an animation display link. If this function is used in a script, you can specify any literal numeric value, analog tagname, or numeric expression.

Example(s)
This script returns 1 if the file c:\data\log.txt exists.

\[ \text{InfoFile("c:\data\log.txt",1,$minute)} \]

This script returns 14223 if the file c:\data\log.txt has a file size of 14223 bytes.

\[ \text{InfoFile("c:\data\log.txt",2,$minute)} \]
This script returns 1138245266 if the file c:\data\log.txt was created on 26th January 2006 at 11:14:26 AM.
```
InfoFile("c:\data\log.txt",3,$minute)
```

This script returns 14 if there are 14 files in the directory c:\data\ that have a txt ending.
```
InfoFile("c:\data\*.txt",4,$minute)
```

**InfoInTouchAppDir() Function**

Returns the current InTouch application directory.

**Syntax**
```
result = InfoInTouchAppDir();
```

**Return Value**
A message tagname to contain the directory of the currently running InTouch application.

**Remarks**
The application directory name may be truncated when passed to a message tagname or shown in an animation link due to the 131 characters limitation.

**Example(s)**
This script may return c:\documents and settings\user1\my documents\my intouch applications\packaging.
```
InfoInTouchAppDir()
```

**InfoResources() Function**

Returns the free bytes of the paging file or the approximate number of Windows tasks.

**Syntax**
```
result = InfoResources (infotype, trigger)
```

**Parameters**

- **infotype**
The type of information you want to retrieve about the Windows environment. A literal integer value, integer tagname, or integer expression with following values and meaning:
  1 - Free bytes of paging file.
  2 - Approximate number of open Windows tasks. This can be used as measurement for the system load.

- **trigger**
  A tagname (or expression) that acts as a trigger to retrieve the system information. If the trigger value changes the system information is retrieved again. A discrete or analog taname or a discrete or analog expression.

**Remarks**
The trigger tag only has meaning when the InfoResources() function is used in an animation display link. If this function is used in a script, you can specify any literal numeric value, analog tagname, or numeric expression.

**Example(s)**
This script retrieves the approximate number of Windows tasks and, if used in an animation display link, updates the information every second.
```
InfoResources(2,$second);
```
**InTouchVersion() Function**

Returns the complete InTouch version number or just parts of it.

**Syntax**

```
result = InTouchVersion (infotype);
```

**Parameters**

- `infotype`
  Specifies how the version information is returned. A literal integer value, integer tagname, or integer expression with the following meaning:
  - 0 - function returns the whole version number
  - 1 - function returns just the major version number
  - 2 - function returns just the minor version number
  - 3 - function returns just the patch level
  - 4 - function returns just the build level

**Example(s)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Possible result</th>
</tr>
</thead>
<tbody>
<tr>
<td>InTouchVersion(0)</td>
<td>10.5.1626.0521.0045.0012</td>
</tr>
<tr>
<td>InTouchVersion(1)</td>
<td>10</td>
</tr>
<tr>
<td>InTouchVersion(2)</td>
<td>5</td>
</tr>
<tr>
<td>InTouchVersion(3)</td>
<td>0</td>
</tr>
<tr>
<td>InTouchVersion(4)</td>
<td>1626</td>
</tr>
</tbody>
</table>

**InvisibleVerifyCredentials() Function**

The InvisibleVerifyCredentials() function can be used in a synchronous QuickScript to verify the credentials of the given user without logging the user on to the InTouch HMI.

**Category**

security

**Syntax**

```
AnalogTag=InvisibleVerifyCredentials( "UserId", "Password", "Domain" );
```

**Arguments**

- `UserId`
  Windows operating system user account name that is part of local computer, workgroup, or domain.

- `Password`
  Password for the account.

- `Domain`
  The Windows domain for the account.
Remarks
If the supplied combination of user, password, and domain are valid then the corresponding access level associated with the user is returned as an integer. Otherwise, -1 is returned.

Note: The InvisibleVerifyCredentials() function must be run from a synchronous QuickScript. The function always returns -1 if run from an asynchronous QuickScript.

This function does not change the currently logged on user. The Domain argument is only valid for operating system-based security. If ArchestrA security is in use and if ArchestrA security is in turn using operating system-based security, the UserId argument should contain the fully qualified user name with domain name or computer name.

Example
AnalogTag=InvisibleVerifyCredentials( "john", "Password", "corporate_hq" );

See Also
PostLogonDialog(), AttemptInvisibleLogon(), IsAssignedRole(), QueryGroupMembership(), AddPermission()

IsAssignedRole() Function
Determines whether the currently logged on user is a member of the specified user role. Only applies to ArchestrA security.

Category
security

Syntax
DiscreteTag=IsAssignedRole( "RoleName" );

Arguments
RoleName
The role associated with an Application Server user.

Remarks
Valid for ArchestrA security mode only and applies to the currently logged on user. If a user is currently logged on and has the RoleName role assigned in the Galaxy IDE, then TRUE is returned. Otherwise, FALSE is returned.

Example
DiscreteTag=IsAssignedRole( "Administrators" );

See Also
AttemptInvisibleLogon(), PostLogonDialog(), InvisibleVerifyCredentials(), QueryGroupMembership(), AddPermission()

LaunchTagViewer() Function
You can start Tag Viewer only when WindowViewer is running, and only after Tag Viewer has been enabled in WindowMaker.

For information about enabling Tag Viewer, see Configuring General WindowViewer Properties in the InTouch® HMI Application Management and Extension Guide.

Syntax
LaunchTagViewer()
**Remarks**

The LaunchTagViewer() function can be executed from any script type except the application scripts OnStartup and OnShutdown.

If Tag Viewer has not been enabled in WindowMaker, calling the function will not start Tag Viewer and a warning message will appear in the logger.

You must have adequate security privileges to start Tag Viewer.

**LogonCurrentUser() Function**

Logs on to InTouch with a user account that is currently logged on to the Windows operating system.

- InTouch configured with OS security: the user is logged on to WindowViewer.
- InTouch configured with ArchestrA security: the user must be a member of ArchestrA OS user-based or OS group-based security.
- InTouch configured with ArchestrA OS user-based or OS group-based security and the user account is configured with smart card credentials: user is logged on using the smart card credentials. The user is logged off if the smart card is removed from the reader.

**Category**

security

**Syntax**

```
IntegerResult = LogonCurrentUser();
```

**Return Value**

Returns -1 and no change to the values assigned to $Operator, $OperatorName, $OperatorDomain, and $AccessLevel if the logon fails.

**Remarks**

This function is available only in InTouch scripting, not in ArchestrA client scripting.

**Example**

```
IntegerResult = LogonCurrentUser();
```

**See Also**

PostLogonDialog(), InvisibleVerifyCredentials(), IsAssignedRole(), AttemptInvisibleLogon(), QueryGroupMembership(), AddPermission()

**PlaySound() Function**

Plays a sound from a wave file or a Windows default sound.

**Syntax**

```
Playsound (soundname, flag)
```

**Parameters**

- `soundname`
  The name of the sound or wave file. A literal string value, message tagname, or string expression. If the sound is defined as a name, it must be defined in the Win.ini file under the [Sounds] section, for example MC="c:\test.wav"

- `flag`
  Specifies how the sound is played. A literal integer value, integer tagname, or integer expression with the following meanings:
  - 0 - Play sound one time synchronously (script execution waits until sound has finished playing).
1 - Play sound one time asynchronously (script execution does not wait until sound has finished playing).
9 - Play sound continuously (until the PlaySound() function is called again).

Example(s)

This script plays the sound of the file c:\welcome.wav one time and holds script execution until it has finished playing.
`PlaySound("c:\welcome.wav",0);`

This script plays the sound Alert continuously. In the win.ini file [Sounds] section you need to associate the sound name Alert with a sound file, such as:
`Alert=c:\alert.wav.
PlaySound("Alert",9);`

PostLogonDialog() Function

Shows the InTouch Logon dialog box and returns TRUE.

Category

security

Syntax

```c
DiscreteTag=PostLogonDialog();
```

Examples

```c
DiscreteTag=PostLogonDialog();
```

See Also

InvisibleVerifyCredentials(), AttemptInvisibleLogon(), IsAssignedRole(), QueryGroupMembership(), AddPermission()

PrintScreen() Function

You can write a script to print the entire WindowViewer screen with the PrintScreen() function.

Syntax

```c
PrintScreen (ScreenOption, PrintOption)
```

Parameters

**ScreenOption**
Determines how much of the WindowViewer screen is to be printed. A literal integer value, integer tagname, or integer expression.

1 - Print the client area, no menus (default)
2 - Print the entire window area, including menus

**PrintOption**
Determines how the printed image is to be stretched to fit on the printout.

- 1 - Best Fit:
  image is stretched so that it fits either horizontally or vertically on the printout without changing the aspect ratio. (default)
- 2 - Vertical Fit:
  image is stretched so that it fits vertically on the printout without changing the aspect ratio. The image may be cut off horizontally.
- 3 - Horizontal Fit:
  image is stretched so that it fits horizontally on the printout without changing the aspect ratio. The image may be cut off vertically.
4 - Stretch to Page:
image is stretched so that it fits horizontally and vertically on the printout. The aspect ratio may change but the image is not truncated.

Invalid options, including 0, default to Best Fit.

**Note:** Popup windows that extend beyond the WindowViewer screen area are cut off.

### Example(s)

This script sends a printout of the current entire WindowViewer screen area without menus to the printer queue. The printout contains the screen area stretched so that it fills the printout dimensions.

```plaintext
PrintScreen(1,4);
```

---

### QueryGroupMembership() Function

Determines whether the currently logged on user is a member of the specified user group. Only applies to operating system security.

**Category**
security

**Syntax**

```plaintext
DiscreteTag=QueryGroupMembership( "Domain", "Group" );
```

**Arguments**

- **Domain**
  Name of the domain or local computer in which the group is located

- **Group**
  Name of the group.

**Remarks**

Valid for operating system security mode only and applies to the currently logged on user. If a user is currently logged on and if he or she is part of the group located on the domain, then TRUE is returned. Otherwise, FALSE is returned.

The QueryGroupMembership() function works with operating system-based security and with ArchestrA security only when the ArchestrA security is set to operating system-based security.

**Examples**

```plaintext
DiscreteTag=QueryGroupMembership( "corporate_hq", "InTouchAdmins" );
DiscreteTag=QueryGroupMembership( "JohnS01", "InTouchUsers" );
```

**See Also**

PostLogonDialog(), InvisibleVerifyCredentials(), IsAssignedRole(), AttemptInvisibleLogon(), AddPermission()

---

### Starting a Windows Application

In a script, you can start a Windows application using the StartApp command.

**Syntax**

```plaintext
StartApp appname;
```

**Parameters**

- **appname**
  Path and file name of the application you want to start. A literal string value, message tagname, or string expression.
Note: You need to know the path and file name of the application. If the application is in a directory that is part of the Windows PATH environment variable, you only need to pass the file name (without path).

Example(s)
This script starts Microsoft Calculator.

StartApp "calc"

SwitchDisplayLanguage() Function
Switches the display of visible, static texts and alarm fields in a desired language for which translated strings are provided.

Category
misc

Syntax
SwitchDisplayLanguage(LocaleID);

Parameter
LocaleID
The language in which static text strings and alarm fields are to be shown at run time.

Example(s)
In this example, German is the language to be shown at run time.
SwitchDisplayLanguage(1031);

See Also
$lang system tag

TseGetClientId() Function
Returns a string version of the client ID (the TCP/IP address of the client) if the View application is running on a Terminal Server client. This client ID is used internally to generate SuiteLink server names and logger file names. Otherwise, the TseGetClientId() function returns an empty string.

Syntax
MessageResult=TseGetClientId();

Example
The client IP address 10.103.202.1 is saved to the MsgTag tag.
MsgTag=TseGetClientId();

TseGetClientNodeName() Function
Returns the client node name if the View application is running on a Terminal Server client assigned a name that can be identified by Windows. Otherwise, the TseGetClientNodeName() function returns an empty string.

Syntax
MessageResult=TseGetClientNodeName();

Example
The client node name is returned as the value assigned to the MsgTag tag.
MsgTag=TseGetClientNodeName();
TseQueryRunningOnClient() Function

Returns a non-zero integer value if the View application is running on a Terminal Services client. Otherwise, it returns a zero.

Syntax

Result = TseQueryRunningOnClient();

Return Value

Returns 0 if View is not running on a Terminal Services client.

Example

IntTag is set to 1 if WindowViewer is running on a Terminal Services client.

IntTag = TseQueryRunningOnClient;

TseQueryRunningOnConsole() Function

The TseQueryRunningOnConsole() function can be run from a script to indicate whether the View application is running on a Terminal Services console.

Syntax

Result = TseQueryRunningOnConsole();

Return Value

Returns a non-zero integer value if the View application is running on a Terminal Services console. Otherwise, the TseQueryRunningOnConsole() function returns a zero.

Example

IntTag is set to 1 if WindowViewer is running on a Terminal Services console.

IntTag = TseQueryRunningOnConsole;

Math Functions

Use math functions to return the answer to the specified mathematical expression.

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

ArcCos()

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

Returns an angle between -1 and 1 (inclusive). The angle is in radians.

### 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()` on page 55, `Sin()` on page 58, `Tan()` on page 59, `ArcSin()` on page 54, `ArcTan()` on page 54

## 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()` on page 55, `Sin()` on page 58, `Tan()` on page 59, `ArcCos()` on page 53, `ArcTan()` on page 54

## 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()` on page 55, `Sin()` on page 58, `Tan()` on page 59, `ArcCos()` on page 53, `ArcSin()` on page 54
Cos()

Returns the cosine of an angle in degrees.

Category
Math

Syntax
\[ \text{Result} = \cos(\text{Number}) ; \]

Parameter
\[ \text{Number} \]
Any number or numeric attribute.

Examples
\begin{align*}
\cos(90); & \ ' \text{ returns 0} \\
\cos(0); & \ ' \text{ returns 1}
\end{align*}

This example shows how to use the function in a math equation:
\[ \text{Wave} = 50 \ast \cos(6 \ast \text{Now().Second}) ; \]

See Also
\begin{itemize}
  \item Sin() on page 58
  \item Tan() on page 59
  \item ArcCos() on page 53
  \item ArcSin() on page 54
  \item ArcTan() on page 54
\end{itemize}

Exp()

Returns the result of the exponent \( e \) raised to a power.

Category
Math

Syntax
\[ \text{Result} = \exp(\text{Number}) ; \]

Parameter
\[ \text{Number} \]
Any number or numeric attribute.

Example
\[ \exp(1); \ ' \text{ returns 2.718...} \]

Int()

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

Category
Math

Syntax
\[ \text{IntegerResult} = \text{Int}(\text{Number}) ; \]

Parameter
\[ \text{Number} \]
Any number or numeric attribute.

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

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

**Category**  
Math

**Syntax**  

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

**Parameter**  

- `Number`  
  Any number or numeric attribute.

**Remarks**  
Natural log of 0 is undefined.

**Examples**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int(4.7);</td>
<td>' returns 4</td>
</tr>
<tr>
<td>Int(-4.7);</td>
<td>' returns -5</td>
</tr>
<tr>
<td>Log(100);</td>
<td>' returns 4.605...</td>
</tr>
<tr>
<td>Log(1);</td>
<td>' returns 0</td>
</tr>
</tbody>
</table>

**See Also**

- LogN() on page 56, Log10() on page 56

Log10()  

Returns the base 10 log of a number.

**Category**  
Math

**Syntax**

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

**Parameter**

- `Number`  
  Any number or numeric attribute.

**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log10(100);</td>
<td>' returns 2</td>
</tr>
</tbody>
</table>

**See Also**

- Log() on page 56, LogN() on page 56

LogN()  

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

**Category**  
Math

**Syntax**

```csharp
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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogN(8, 3);</td>
<td>' returns 1.89279</td>
</tr>
<tr>
<td>LogN(3, 7);</td>
<td>' returns 0.564</td>
</tr>
</tbody>
</table>

See Also

Log() on page 56, Log10() on page 56

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round(4.3, 1);</td>
<td>' returns 4</td>
</tr>
<tr>
<td>Round(4.3, .01);</td>
<td>' returns 4.30</td>
</tr>
<tr>
<td>Round(4.5, 1);</td>
<td>' returns 5</td>
</tr>
<tr>
<td>Round(-4.5, 1);</td>
<td>' returns -4</td>
</tr>
<tr>
<td>Round(106, 5);</td>
<td>' returns 105</td>
</tr>
<tr>
<td>Round(43.7, .5);</td>
<td>' returns 43.5</td>
</tr>
</tbody>
</table>

See Also

Trunc() on page 59
**Sgn()**

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

**Category**

Math

**Syntax**

\[
\text{IntegerResult} = \text{Sgn( Number )};
\]

**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;

---

**Sin()**

Returns the sine of an angle in degrees.

**Category**

Math

**Syntax**

\[
\text{Result} = \text{Sin( Number )};
\]

**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:

\[
\text{wave} = 100 \times \text{Sin}(6 \times \text{Now().Second})
\]

**See Also**

`Cos()` on page 55, `Tan()` on page 59, `ArcCos()` on page 53, `ArcSin()` on page 54, `ArcTan()` on page 54

---

**Sqrt()**

Returns the square root of a number.

**Category**

Math

**Syntax**

\[
\text{RealResult} = \text{Sqrt( Number )};
\]
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:

\[ x = \sqrt{me.PV}; \]

Tan()

Returns the tangent of an angle given in degrees.

Category

Math

Syntax

\[ \text{Result} = \text{Tan}( \text{Number} ); \]

Parameter

Number
The angle in degrees. Any number or numeric attribute.

Examples

\[ \text{Tan}(45); \quad ' \text{returns} \ 1; \]
\[ \text{Tan}(0); \quad ' \text{returns} \ 0; \]

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

\[ \text{Wave} = 10 + 50 \times \text{Tan}(6 \times \text{Now()}.\text{Second}); \]

See Also

Cos() on page 55, Sin() on page 58, ArcCos() on page 53, ArcSin() on page 54, ArcTan() on page 54

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

\[ \text{NumericResult} = \text{Trunc}( \text{Number} ); \]

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

\[ \text{Trunc}(4.3); \quad ' \text{returns} \ 4; \]
\[ \text{Trunc}(-4.3); \quad ' \text{returns} \ -4; \]

See Also

Round() on page 57
Miscellaneous Functions

Functions in the miscellaneous group perform a variety of purposes, such as logging data or querying attributes.

ActivateApp()

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

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.

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

**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);``` 

**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() on page 62, IsInitializing() on page 62, IsUncertain() on page 62, IsUsable() on page 63

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() on page 61, IsInitializing() on page 62, IsUncertain() on page 62, IsUsable() on page 63

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() on page 61, IsGood() on page 62, IsUncertain() on page 62, IsUsable() on page 63

IsUncertain()

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

**Syntax**

```csharp
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**

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

**See Also**

`IsBad()` on page 61, `IsGood()` on page 62, `IsInitializing()` on page 62, `IsUsable()` on page 63

**IsUsable()**

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

**Category**
Miscellaneous

**Syntax**

```csharp
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**

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

**See Also**

`IsBad()` on page 61, `IsGood()` on page 62, `IsInitializing()` on page 62, `IsUncertain()` on page 62

**LogCustom()**

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

**Category**
Miscellaneous
### Syntax

```csharp
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 LogCustom 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

```csharp
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.
```

### 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

```csharp
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);

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

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

The above statement writes the following to the Log Viewer:

```
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

```plaintext
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

```plaintext
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

```plaintext
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.

**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.

**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:

<table>
<thead>
<tr>
<th>Key</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKSPACE</td>
<td>{BACKSPACE} or {BS}</td>
</tr>
<tr>
<td>BREAK</td>
<td>{BREAK}</td>
</tr>
<tr>
<td>CAPSLOCK</td>
<td>{CAPSLOCK}</td>
</tr>
<tr>
<td>DELETE</td>
<td>{DELETE} or {DEL}</td>
</tr>
<tr>
<td>DOWN</td>
<td>{DOWN}</td>
</tr>
</tbody>
</table>
Key | Code
--- | ---
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:

Key | Code
--- | ---
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. For buffered values, only the last calculated value is captured for historization.

**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**

Interim calculated buffered values are NOT historized. Use SetAttributeVT2() if historization of interim values is needed.

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);
```

**SetAttributeVT2()**

Sets the value and timestamp of an object attribute. This function is identical to SetAttributeVT(), but SetAttributeVT2() allows interim calculated data for buffered values to be historized once per scan cycle.

**Category**
Miscellaneous

**Syntax**

```
SetAttributeVT2( 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**

In contrast to SetAttributeVT(), SetAttributeVT2() allows historization of interim calculated buffered values.

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 SetAttributeVT2() 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 (interim calculated values for buffered data are historized).

```
SetAttributeVT2(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() on page 70, SetInitializing() on page 71, SetUncertain() on page 71

### 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() on page 70, SetInitializing() on page 71, SetUncertain() on page 71

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() on page 70, SetGood() on page 70, SetUncertain() on page 71

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() on page 70, SetGood() on page 70, SetInitializing() on page 71
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

```csharp
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 ArchestrA 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:

```csharp
```

uses the function as follows:

```csharp
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.

**Title_Bar_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 the topic *Signature Security for Acknowledging Alarms*, under “Adding and Maintaining Symbol Scripts” in the *Creating and Managing ArchestrA Graphics User Guide*.

**Examples**

```vbnet
Dim n as Integer;
```

Using an array of strings:

```vbnet
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, "Aking 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**

```vbnet
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 the topic *Examples of Using the Attribute Parameter in the SignedWrite() Function* under "Managing Symbols" in the *Creating and Managing ArchestrA Graphics User 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":
```vbscript
Dim strselect As String;
Dim x As Variant;
{ logic to set strselect to "A" or "B" }
x.Indirect("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 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.
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.

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.

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.

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.

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.

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

For more information about using the SignedWrite() function, see the topic Working with the SignedWrite() Function for Secured and Verified Writes under "Managing Symbols" in the Creating and Managing ArchestrA Graphics User Guide.

**Examples**

```csharp
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:

```csharp
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.

### WriteStatus()

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

**Category**

Miscellaneous

**Syntax**

```csharp
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**

```csharp
WriteStatus(TIC101.SP);
```
**WWControl()**

Restores, minimizes, maximizes, or closes an application.

**Category**

Miscellaneous

**Syntax**

\[ \text{WWControl( AppTitle, ControlType );} \]

**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**

\[ \text{WWControl("Calculator","Restore");} \]

**See Also**

ActivateApp() on page 60

**String Functions**

Use string functions to work with character strings and string values.

**DText()**

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

**Category**

String

**Syntax**

\[ \text{StringResult }= \text{DText( Discrete, OnMsg, OffMsg );} \]

**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**

\[ \text{StringResult }= \text{DText(me.temp > 150, "Too hot", "Just right");} \]
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() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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**

```
StringCompareNoCase( 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.
StringFromIntg

Syntax

\[ \text{StringResult} = \text{StringFromIntg}( \text{Number}, \text{numberBase} ); \]

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() on page 81, StringChar() on page 81, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

StringFromReal

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

Syntax

\[ \text{StringResult} = \text{StringFromReal}( \text{Number, Precision, Type} ); \]

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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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()` on page 81, `StringChar()` on page 81, `StringFromIntg()` on page 83, `StringFromReal()` on page 84, `StringFromTime()` on page 85, `StringLeft()` on page 87, `StringLen()` on page 88, `StringLower()` on page 88, `StringMid()` on page 89, `StringReplace()` on page 89, `StringRight()` on page 90, `StringSpace()` on page 91, `StringTest()` on page 91, `StringToIntg()` on page 92, `StringToReal()` on page 93, `StringTrim()` on page 93, `StringUpper()` on page 94, `Text()` on page 95

**StringLeft()**

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

**Category**
String

**Syntax**

```plaintext
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()` on page 81, `StringChar()` on page 81, `StringFromIntg()` on page 83, `StringFromReal()` on page 84, `StringFromTime()` on page 85, `StringInString()` on page 86, `StringLen()` on page 88, `StringLower()` on page 88, `StringMid()` on page 89, `StringReplace()` on page 89, `StringRight()` on page 90, `StringSpace()` on page 91, `StringTest()` on page 91, `StringToIntg()` on page 92, `StringToReal()` on page 93, `StringTrim()` on page 93, `StringUpper()` on page 94, `Text()` on page 95
**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()` on page 81,
- `StringChar()` on page 81,
- `StringFromIntg()` on page 83,
- `StringFromReal()` on page 84,
- `StringFromTime()` on page 85,
- `StringInString()` on page 86,
- `StringLeft()` on page 87,
- `StringLowerCase()` on page 88,
- `StringMid()` on page 89,
- `StringReplace()` on page 89,
- `StringRight()` on page 90,
- `StringSpace()` on page 91,
- `StringTest()` on page 91,
- `StringToUpper()` on page 92,
- `Text()` on page 95

**StringLowerCase()**

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

**Category**
String

**Syntax**

```
StringResult = StringLowerCase( 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**

- `StringLowerCase("TURBINE")` returns "turbine";
- `StringLowerCase("22.2 Is The Value")` returns "22.2 is the value";
See Also
StringASCII() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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() on page 87 function and StringRight() on page 90 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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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() on page 89 function does not recognize special characters, such as @ # $ % & * (). It reads them as delimiters. For example, if the function StringReplace() on page 89 (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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringSpace() on page 91, StringTest() on page 91, StringToIn()g() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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 "×" character:
StringSpace(4)  ' returns "××××"
"Pump" + StringSpace(1) + "Station"  ' returns "Pump×Station"
```

See Also
StringASCII() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringTest() on page 91, StringToIn()g() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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() on page 91 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() on page 91 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() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, StringUpper() on page 94, Text() on page 95

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

```plaintext
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()](#) on page 81, [StringChar()](#) on page 81, [StringFromIntg()](#) on page 83, [StringFromReal()](#) on page 84, [StringFromTime()](#) on page 85, [StringInString()](#) on page 86, [StringLeft()](#) on page 87, [StringLen()](#) on page 88, [StringLower()](#) on page 88, [StringMid()](#) on page 89, [StringReplace()](#) on page 89, [StringRight()](#) on page 90, [StringSpace()](#) on page 91, [StringTest()](#) on page 91, [StringToIntg()](#) on page 92, [StringTrim()](#) on page 93, [StringUpper()](#) on page 94, [Text()](#) on page 95

**StringToReal()**

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

**Category**

String

**Syntax**

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

**Parameter**

<table>
<thead>
<tr>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>String that function acts on. Actual string or a string attribute.</td>
</tr>
</tbody>
</table>

**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**

```plaintext
StringToReal("ABCD"); ' returns 0;
StringToReal("22.261 is the value"); ' returns 22.261;
StringToReal("The Value is 2"); ' returns 0;
```

See Also

[StringASCII()](#) on page 81, [StringChar()](#) on page 81, [StringFromIntg()](#) on page 83, [StringFromReal()](#) on page 84, [StringFromTime()](#) on page 85, [StringInString()](#) on page 86, [StringLeft()](#) on page 87, [StringLen()](#) on page 88, [StringLower()](#) on page 88, [StringMid()](#) on page 89, [StringReplace()](#) on page 89, [StringRight()](#) on page 90, [StringSpace()](#) on page 91, [StringTest()](#) on page 91, [StringToIntg()](#) on page 92, [StringTrim()](#) on page 93, [StringUpper()](#) on page 94, [Text()](#) on page 95

**StringTrim()**

Removes unwanted spaces from text and returns the result.

**Category**

String
Syntax

```plaintext
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 "×" character.

```plaintext
StringTrim("×××××This×is×a××test×××××", 1) ' returns "This×is×a××test×××××";
StringTrim("×××××This×is×a××test×××××", 2) ' returns "×××××This×is×a××test";
StringTrim("×××××This×is×a××test×××××", 3) ' returns "This×is×a×test";
```

The `StringReplace()` on page 89 function can remove ALL spaces from a specified a string attribute. Simply replace all the space characters with a "null."

See Also

- `StringASCII()` on page 81
- `StringChar()` on page 81
- `StringFromIntg()` on page 83
- `StringFromReal()` on page 84
- `StringFromTime()` on page 85
- `StringInString()` on page 86
- `StringLeft()` on page 87
- `StringLen()` on page 88
- `StringLower()` on page 88
- `StringMid()` on page 89
- `StringReplace()` on page 89
- `StringRight()` on page 90
- `StringSpace()` on page 91
- `StringTest()` on page 91
- `StringToReal()` on page 93
- `StringUpper()` on page 94
- `Text()` on page 95

**StringUpper()**

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

Category

String

Syntax

```plaintext
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

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

StringASCII() on page 81, StringChar() on page 81, StringFromIntg() on page 83, StringFromReal() on page 84, StringFromTime() on page 85, StringInString() on page 86, StringLeft() on page 87, StringLen() on page 88, StringLower() on page 88, StringMid() on page 89, StringReplace() on page 89, StringRight() on page 90, StringSpace() on page 91, StringTest() on page 91, StringToIntg() on page 92, StringToReal() on page 93, StringTrim() on page 93, Text() on page 95

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, ",".00"));

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

MessageTag = "One + " + Text(1,"".00") + StringChar(32) + "Two +" + Text(2,"".00");

See Also

StringFromIntg() on page 83, StringToIntg() on page 92, StringFromReal() on page 84, StringToReal() on page 93

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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$wwStringFromTime(57600, 1)</td>
<td>1/2/70</td>
</tr>
<tr>
<td>$wwStringFromTime(57600, 2)</td>
<td>12:00:00 AM</td>
</tr>
<tr>
<td>$wwStringFromTime(57600, 3)</td>
<td>Fri Jan 02 00:00:00 1970</td>
</tr>
<tr>
<td>$wwStringFromTime(57600, 4)</td>
<td>Fri</td>
</tr>
<tr>
<td>$wwStringFromTime(57600, 5)</td>
<td>Friday</td>
</tr>
</tbody>
</table>

System Functions

Use system functions to interact with the operating system or other core system functions, such as ActiveX objects.

CreateObject()

Creates an ActiveX (COM) object.

Category
System

Syntax

\[ \text{ObjectResult} = \text{CreateObject}( \text{ProgID} ); \]

Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgID</td>
<td>The program ID (as a string) of the object to be created.</td>
</tr>
</tbody>
</table>

Example

\[ \text{CreateObject("ADODB.Connection")}; \]

Now()

Returns the current time.

Category
System

Syntax

\[ \text{TimeValue} = \text{Now}(); \]
Remarks
The return value can be formatted using .NET functions.

WWDDE Functions
Use WWDDE functions when working with the DDE protocol.

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 WWExecute() 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 WWDDE 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 WWExecute() 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="Macrol!TestMacro";
Command="[Run(" + StringChar(34) + Macro + StringChar(34) + ",0))];"
WWExecute("excel","system",Command);
```

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

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

The following script executes a macro in Microsoft Access:

```
WWExecute("MSAccess","system","MyMacro");
```
**WW Poke()**

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

**Category**

WWDDE

**Syntax**

\[
\text{Status} = \text{WWPoke}(\text{Application}, \text{Topic}, \text{Item}, \text{TextValue});
\]

**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()` on page 95, `StringFromIntg()` on page 83, or `StringFromReal()` on page 84 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 WW Request() 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:

```plaintext
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()` on page 95, `StringFromIntg()` on page 83, `StringFromReal()` on page 84

**WW Request()**

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

Syntax

\[
\text{Status} = \text{WWRequest( Application, Topic, Item, Attribute );}
\]

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 WWDE 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()** on page 92 or **StringToReal()** on page 93 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:

\[
\text{WWRequest("excel","[Book1.xls]sheet1","r1c1",Result);} \\
\text{Value=StringToReal(Result);} \\
\]

See Also

**StringToIntg()** on page 92, **StringToReal()** on page 93

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:

<table>
<thead>
<tr>
<th>DIM</th>
<th>Required keyword.</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable_name&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>&lt;upper_bound&gt;</td>
<td>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.</td>
</tr>
<tr>
<td>AS</td>
<td>Optional keyword for declaring the variable’s datatype.</td>
</tr>
<tr>
<td>&lt;data_type&gt;</td>
<td>Any one of the following 11 datatypes: Boolean, Discrete, Integer, ElapsedTime, Float, Real, Double, String, Message, Time or Object. Data_type can also be a .NET data_type like System.Xml.XmlDocument or a type defined in an imported script library.</td>
</tr>
</tbody>
</table>

If you omit the AS clause from the DIM statement, the variable, by default, is declared as an Integer datatype. For example:

```
DIM LocVar1;
```

is equivalent to:

```
DIM LocVar1 AS Integer;
```

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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
FloatConst ::= [<sign>] <digit>* .<digit>+[<exponent>;]
```
or

\[ [\text{<sign>}] \space \text{<digit>}+ [\text{.<digit>\ast} \ [\text{<exponent>}]}; \]

where:

- \text{sign} ::= + or -
- \text{digit} ::= 0-9 (can be one or more decimal digits)
- \text{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:

\text{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:

\text{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:

- \text{IF ... THEN ... ELSEIF ... ELSE ... ENDIF} on page 102
- \text{FOR ... TO ... STEP ... NEXT} Loop on page 104
- \text{FOR EACH ... IN ... NEXT} on page 105
- \text{TRY ... CATCH} on page 106
- \text{WHILE} Loop on page 107

\text{IF ... THEN ... ELSEIF ... ELSE ... ENDIF}

\text{IF-THEN-ELSE-ENDIF} conditionally executes various instructions based on the state of an expression. The syntax is as follows:

\text{IF} \ <\text{Boolean} \_\text{expression}> \ \text{THEN}
\hspace{1cm} [\text{statements}];
\hspace{1cm} [ \{ \text{ELSEIF} \]
\hspace{2cm} [\text{statements} } \} ];
\hspace{1cm} [ \text{ELSE} \]
\hspace{2cm} [\text{statements} ];
\hspace{1cm} \text{ENDIF};

Where \text{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:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean, Discrete</td>
<td>Directly used (no mapping needed).</td>
</tr>
</tbody>
</table>
## Data Type Mapping

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Value = 0 evaluated as False. Value != 0 evaluated as True.</td>
</tr>
<tr>
<td>Float, Real</td>
<td>Value = 0 evaluated as False. Value != 0 evaluated as True.</td>
</tr>
<tr>
<td>Double</td>
<td>Value = 0 evaluated as False. Value != 0 evaluated as True.</td>
</tr>
<tr>
<td>String, Message</td>
<td>Cannot be mapped. Using an expression that results in a string type as the Boolean_expression results in a script validation error.</td>
</tr>
<tr>
<td>Time</td>
<td>Cannot be mapped. Using an expression that results in a time type as the Boolean_expression results in a script validation error.</td>
</tr>
<tr>
<td>ElapsedTime</td>
<td>Cannot be mapped. Using an expression that results in an elapsed time type as the Boolean_expression results in a script validation error.</td>
</tr>
<tr>
<td>Object</td>
<td>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.</td>
</tr>
</tbody>
</table>

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. For example:

```plaintext
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 approach nests a second IF compound statement within a previous one and requires an additional ENDIF:

```plaintext
IF (X1 == 1) THEN
  X1 = 5;
  { ELSEIF <X1 == 2> THEN
    X1 = 10;
    ELSEIF X1 == 3 THEN
      X1 = 20;
      ELSEIF X1 == 4 THEN
        X1 = 30;
        IF X1 == 99 THEN
          X1 = 0;
        ENDIF;
      ENDIF;
    ELSEIF X1 == 5 THEN
      X1 = 15;
    ENDIF;
  ENDIF;
ENDIF;
```
See "Sample Scripts" 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 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.Attr2 = 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.

---

**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:

```plaintext
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.
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:**

```vbnet
dim command = new System.Data.SqlClient.SqlCommand;
dim reader as System.Data.SqlClient.SqlDataReader;
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;
    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.

**QuickScript .NET Operators**

The following QuickScript .NET operators require a single operand:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Complement</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>

The following QuickScript .NET operators require two operands:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition and concatenation</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
</tbody>
</table>
Operator | Short Description
---|---
** | 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

The following table shows the precedence of QuickScript .NET operators:

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (highest)</td>
<td>( )</td>
</tr>
<tr>
<td>2</td>
<td>- (negation), NOT, ~</td>
</tr>
<tr>
<td>3</td>
<td>**</td>
</tr>
<tr>
<td>4</td>
<td>*, /, MOD</td>
</tr>
<tr>
<td>5</td>
<td>+, - (subtraction)</td>
</tr>
<tr>
<td>6</td>
<td>SHL, SHR</td>
</tr>
<tr>
<td>7</td>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td>8</td>
<td>==, &lt;&gt;</td>
</tr>
<tr>
<td>9</td>
<td>&amp;</td>
</tr>
<tr>
<td>10</td>
<td>^</td>
</tr>
</tbody>
</table>
Precedence | Operator
---|---
11 | |
12 =
13 AND
14 (lowest) OR

The arguments of the listed operators can be numbers or attribute values. Putting parentheses around an argument is optional. 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:

```plaintext
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>97 MOD 8</td>
<td>yields 1</td>
</tr>
<tr>
<td>63 MOD 5</td>
<td>yields 3</td>
</tr>
</tbody>
</table>

### 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:

```plaintext
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ^ 0</td>
<td>yields 0</td>
</tr>
<tr>
<td>0 ^ 1</td>
<td>yields 1</td>
</tr>
<tr>
<td>1 ^ 0</td>
<td>yields 1</td>
</tr>
<tr>
<td>1 ^ 1</td>
<td>yields 0</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 yields 0</td>
</tr>
<tr>
<td>0</td>
<td>1 yields 1</td>
</tr>
<tr>
<td>1</td>
<td>0 yields 1</td>
</tr>
<tr>
<td>1</td>
<td>1 yields 1</td>
</tr>
</tbody>
</table>

### 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 understand the QuickScript .NET scripting language.

**Important Notes:** 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.

Some sample scripts include references to public websites as examples. You may need to replace those URLs with a current and verified URLs.

### In This Chapter

- Accessing an Excel Spreadsheet Using an Imported Type Library ................................................................. 113
- Accessing an Excel Spreadsheet Using CreateObject .......................................................................................... 114
- Calling a Web Service to Get the Temperature for a Specified Zip Code .......................................................... 114
- Calling a Web Service to Send an E-mail Message .............................................................................................. 115
- Creating a Look-up Table and Doing a Look-up on It ............................................................................................ 115
- Creating an XML Document and Saving it to Disk ............................................................................................... 115
- Executing a SQL Parameterized INSERT Command .......................................................................................... 116
- Filling a String Array and Using It ....................................................................................................................... 116
- Filling a Two-Dimensional Integer Array and Using It ......................................................................................... 117
- Formatting a Number Using a .NET Format ‘Picture’ .......................................................................................... 117
- Formatting a Time Using a .NET Format ‘Picture’ .............................................................................................. 117
- Getting the Directories Under the C Drive ........................................................................................................... 117
- Loading an XML Document from Disk and Doing Look-ups on It ...................................................................... 117
- Querying a SQL Server Database ....................................................................................................................... 117
- Reading a Performance Counter ......................................................................................................................... 118
- Reading a Text File from Disk ............................................................................................................................. 118
- Sharing a SQL Connection or Any Other .NET Object ....................................................................................... 118
- Using DDE to Access an Excel Spreadsheet ..................................................................................................... 119
- Using Microsoft Exchange to Send an E-mail Message ...................................................................................... 119
- Using Screen-Scraping to Get the Temperature for a City .................................................................................. 119
- Using SMTP to Send an E-mail Message ............................................................................................................. 120
- Writing a Text File to Disk ................................................................................................................................... 120
- Dynamically Binding an Indirect Variable to a Reference .................................................................................. 120

## Accessing an Excel Spreadsheet Using an Imported Type Library

The purpose of this script is to write data to an open Excel spreadsheet.

Before using this script, you must first:

1. Import the Microsoft Office Excel dll (**Microsoft.Office.Interop.Excel.dll**) to create the required namespace. From the **Galaxy** menu, select **Import**, then **Script Function Library**.
Note: The Microsoft Office Excel dll file name and location may vary, depending on which version of Excel is installed.

2. Open the Excel spreadsheet you want to access. In the following sample script, the Excel file name and path are: C:\documents\sample.xlsx

```vbscript
dim app as object;
dim wb as object;
dim ws as object;

app = CreateObject("Excel.Application");
app.Visible=true;
wb = app.Workbooks.Open("C:\documents\sample.xlsx");
ws = wb.ActiveSheet;
ws.Range("A1").Value = 100;
ws.Range("A2").Value = 200;
ws.Range("A3").Value = ";=A1*A2"
LogMessage(ws.Range("A3").Value);
wb.Visible=true;
wb.Close(false);
```

Accessing an Excel Spreadsheet Using CreateObject

```vbscript
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);
```

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);
doc.SetAttribute("isbn", "0385503822");
```csharp
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

```csharp
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;database=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

```csharp
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

```vbs
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'

```vbs
dim i as integer;
i = 1234;
LogMessage("Total cost: " + i.ToString("$#,###,###.00").
```

Formatting a Time Using a .NET Format 'Picture'

```vbs
dim t as time;
t = Now();
LogMessage("The current time is: " + t.ToString("hh:mm:ss") + ".");
```

Getting the Directories Under the C Drive

```vbs
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

```vbs
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

```vbs
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;database=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;database=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;
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:

```
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

```csharp
{
    {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

```csharp
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:

```csharp
' 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 ArchestrA 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:

```csharp
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.
Index

A
Abs() • 53
Accepting Autocomplete Suggestions • 14
Accessing an Excel Spreadsheet Using an Imported Type Library • 113
Accessing an Excel Spreadsheet Using CreateObject • 114
ActivateApp() • 60
AddPermission() Function • 35
and system resources • 9
AND, OR, and NOT • 111
ArcCos() • 53
ArcSin() • 54
ArcTan() • 54
Assignment (=) • 110
AttemptInvisibleLogon() Function • 36
Autocomplete • 10

B
Bitwise AND (&) • 110

C
Calling a Web Service to Get the Temperature for a Specified Zip Code • 114
Calling a Web Service to Send an E-mail Message • 115
ChangePassword() Function • 36
Closing a Client Application Window • 17
closing a client application window containing scripts • 17
Color Indicators for Script Elements • 10
Common Scripting Environment • 7
Comparisons (<, >, <=, >=, ==, <> ) • 111
Complement (~) • 109
Contact Information • 2
converting • 109
Cos() • 55
CreateObject() • 96

Creating a Look-up Table and Doing a Look-up on It • 115
Creating an XML Document and Saving it to Disk • 115

D
DateTimeGMT() • 61
Deployment Scripts • 9
deployment timeout period • 9
deployment timeout period, scripts • 9
description • 17
DText() • 80
dynamic reference scripting • 14
dynamic references • 14
dynamic referencing • 14
Dynamic Referencing Considerations • 14
Dynamically Binding an Indirect Variable to a Reference • 120

E
EnableDisableKeys() Function • 37
entries • 7
errors • 14
Exclusive OR (^) and Inclusive OR ( | ) • 110
Execute method • 14
Execute Scripts • 8
Executing a SQL Parameterized INSERT Command • 116
Exp() • 55

F
FileCopy() Function • 38
FileDelete() Function • 39
FileMove() Function • 39
FileReadFields() Function • 40
FileReadMessage() Function • 41
FileWriteFields() Function • 41
FileWriteMessage() Function • 42
Filling a String Array and Using It • 116
Filling a Two-Dimensional Integer Array and Using It • 117
Filtering Events • 60
FOR ... TO ... STEP ... NEXT Loop • 104
FOR EACH ... IN ... NEXT • 105
Formatting a Number Using a .NET Format 'Picture' • 117
Formatting a Time Using a .NET Format 'Picture' • 117
GetAccountStatus() Function • 43
GetCPQuality() • 19
GetCPTimeStamp() • 20
GetNodeName() Function • 43
Getting the Directories Under the C Drive • 117
Graphic Client Functions • 19
HideGraphic() • 20
HideSelf() • 21
IF ... THEN ... ELSEIF ... ELSE ... ENDIF • 102
IF ... THEN ... ELSEIF ... ELSE ... ENDIF and Attribute Quality • 104
InfoAppTitle() Function • 44
InfoDisk() Function • 44
InfoFile() Function • 45
InfoInTouchAppDir() Function • 46
InfoResources() Function • 46
Int() • 55
InTouch Functions • 35
InTouchVersion() Function • 47
InvisibleVerifyCredentials() Function • 47
IsAssignedRole() Function • 48
IsBad() • 61
IsGood() • 62
IsInitializing() • 62
IsUncertain() • 62
IsUsable() • 63
LaunchTagViewer() Function • 48
Line Numbers • 18
Loading an XML Document from Disk and Doing Look-ups on It • 117
Log Functions • 18
Log() • 56
Log10() • 56
LogCustom() • 63
LogDataChangeEvent() • 64
LogDataChangeEvent() script function • 64
LogError() • 65
LogMessage() • 65
LogN() • 56
Logoff() • 21
LogonCurrentUser() Function • 49
LogTrace() • 66
LogWarning() • 66
Math Functions • 53
Maximizing or Restoring a Client Application Window • 17
memory load and scripts • 9
memory load during deployment • 9
memory types • 109
message types • 109
message types, concatenating • 109
Minimizing a Client Application Window • 17
minimizing a client application window containing scripts • 17
Miscellaneous Functions • 60
Modulo (MOD) • 110
Multi-level Undo and Redo • 14
Multiplication ( * ), Division ( / ), Addition ( + ), Subtraction ( - ) • 109
Negation ( - ) • 109
Now() • 96
Numbers and Strings • 101
OffScan Scripts • 9
OnScan Scripts • 8
OnScan scripts system resources • 9
Opening a Client Application Window • 17
opening a client application window containing scripts • 17
order of evaluation, operators • 109

P
Parentheses ( ) • 109
Pi() • 57
PlaySound() Function • 49
positive integers • 109
PostLogonDialog() Function • 50
Power ( ** ) • 109
PrintScreen() Function • 50

Q
QueryGroupMembership() Function • 51
Querying a SQL Server Database • 117
QuickScript .NET Control Structures • 102
QuickScript .NET Functions • 19
QuickScript .NET Operators • 107
QuickScript .NET Variables • 99

R
raising to the power • 109
Reading a Performance Counter • 118
Reading a Text File from Disk • 118
real numbers • 109
reference scripting • 14
referencing • 14
Required Syntax for Expressions and Scripts • 7
resource load and scripts • 9
resources and deployment • 9
restoring a client application window containing scripts • 17
Round() • 57
Run-Time Client Script Behavior • 17

S
Sample QuickScript .NET Scripts • 113
Script Editing Styles and Syntax • 7
Script Execution Types • 8
Script Functions • 19
SendKeys() • 67
SetAttributeVT() • 69
SetAttributeVT() script function • 69
SetAttributeVT2() • 69
SetBad() • 70
SetGood() • 70
SetInitializing() • 71
SetUncertain() • 71
Sgn() • 58
Sharing a SQL Connection or Any Other .NET Object • 118
Shift Left (SHL), Shift Right (SHR) • 110
ShowContent() • 22
ShowGraphic() • 25
ShowLoginDialog() • 34
Shutdown Scripts • 9
SignedAlarmAck() • 72
SignedWrite() • 75
Simple Scripts • 8
Sin() • 58
Sqrt() • 58
Starting a Windows Application • 51
Startup Scripts • 8
String Functions • 80
StringASCII() • 81
StringChar() • 81
StringCompare() • 82
StringCompareNoCase() • 82
StringFromGMTTimeToLocal() • 83
StringFromIntg() • 83
StringFromReal() • 84
StringFromTime() • 85
StringFromTimeLocal() • 85
StringInString() • 86
StringLeft() • 87
StringLength() • 88
StringLower() • 88
StringMid() • 89
StringReplace() • 88
StringRight() • 89
StringLength() • 91
StringTest() • 91
StringToIntg() • 92
StringToReal() • 93
StringTrim() • 93
StringUpper() • 94
SwitchDisplayLanguage() Function • 52
System Functions • 96
system resources, startup scripts • 9
T
Tan() • 59
Text() • 95
timestamps • 64, 69
Trunc() • 59
TRY ... CATCH • 106
TseGetClientId() Function • 52
TseGetClientNodeName() Function • 52
TseQueryRunningOnClient() Function • 53
TseQueryRunningOnConsole() Function • 53
Types category • 19
Types category, script function • 19

U
Using DDE to Access an Excel Spreadsheet • 119
Using Microsoft Exchange to Send an E-mail Message • 119
Using Screen-Scraping to Get the Temperature for a City • 119
Using SMTP to Send an E-mail Message • 120

V
Vista security restrictions • 60
Visual Indication of Script Errors • 18

W
WHILE Loop • 107
Working with QuickScript Editor Features • 10
WriteStatus() • 79
Writing a Text File to Disk • 120
WWControl() • 80
WWDDE Functions • 97
WWExecute() • 97
WWPoke() • 98
WWRequest() • 98
WWStringFromTime() • 95