Contents

Chapter 1  Introduction to Scripting ............................................................................. 11
  About Scripting ........................................................................................................ 11
  Basic Scripting Concepts ....................................................................................... 11
    Types of Scripts .................................................................................................. 11
  Advanced Scripting Concepts ................................................................................ 12
    OLE Objects ......................................................................................................... 12
    Scripting with ActiveX Controls ....................................................................... 12

Chapter 2  Creating and Editing Scripts .................................................................... 13
  About Creating and Editing Scripts ...................................................................... 13
  Working with the InTouch Script Editor ................................................................. 14
    Color Indicators for InTouch Script Elements .................................................. 15
    InTouch Script Editor Autocomplete Features ............................................... 15
    Accepting InTouch Script Editor Autocomplete Suggestions ....................... 16
    Multi-level Undo and Redo in InTouch Scripting .............................................. 16
    Visual Indication of InTouch Script Errors ...................................................... 17
    Managing InTouch Script Edits ......................................................................... 17
  Opening a Script for Editing ................................................................................ 17
  Saving or Discarding Changes to a Script ............................................................. 18
  Copying, Cutting and Pasting Text ......................................................................... 19
  Searching Within a Script ..................................................................................... 19
    Configuring a Find or Replace .......................................................................... 21
    Search by Regular Expressions .......................................................................... 22
    Search by Wildcard Characters ......................................................................... 24
    Search by Acronym ........................................................................................... 25
    Search by Shorthand .......................................................................................... 26
  Inserting Code Elements ....................................................................................... 26
  Accessing Help for Script Functions .................................................................... 27
  Validating Scripts for Correct Syntax .................................................................. 27
  Printing Scripts ...................................................................................................... 27
  Deleting Scripts ..................................................................................................... 28

Chapter 3  Script Triggers .......................................................................................... 29
  About Script Triggers ............................................................................................. 29
  Types of Script Triggers ......................................................................................... 30
  Using Multiple Triggers ......................................................................................... 30
  Periodic Script Execution ..................................................................................... 30
  Configuring Application Scripts .......................................................................... 30
## Chapter 4  The Script Language ................................................................. 43

### Contents

- **Limitations of Application Scripts** .......................................................... 31
- **Configuring Window Scripts** ................................................................... 32
- **Configuring Key Scripts** .......................................................................... 33
- **Configuring Condition Scripts** .................................................................. 34
- **Configuring Data Change Scripts** ............................................................ 36
- **Configuring Action Scripts** ....................................................................... 36
- **Configuring ActiveX Event Scripts** .......................................................... 39
- **Pausing Script Execution at Run Time** ...................................................... 41
- **$LogicRunning System Tag** ...................................................................... 42

**Chapter 4**  The Script Language ................................................................ 43

**About the Script Language** ......................................................................... 43

**Basic Syntax Rules** .................................................................................... 43

- **Subroutines** ............................................................................................... 44
- **Statements** ................................................................................................. 44
- **Indentation** ................................................................................................. 44
- **Comments** .................................................................................................. 44
- **Tag References** ........................................................................................... 44
- **Literal Data Values** ...................................................................................... 44
- **Value Expressions** ...................................................................................... 44
- **Syntax Validation** ........................................................................................ 44

**Calling Standard Functions** ........................................................................ 45

- **Syntax for Calling Standard Functions** ...................................................... 45
- **Passing Parameters to a Function** .............................................................. 45

**Calling Custom Functions (QuickFunctions)** ............................................... 45

- **Passing Parameters to a QuickFunction** ..................................................... 45

**Value Assignments and Operators** ............................................................. 46

- **Supported Operators** ................................................................................ 46
  - Addition or Concatenation: + ..................................................................... 47
  - Subtraction: - ............................................................................................ 48
  - Multiplication: * ......................................................................................... 48
  - Division: / ................................................................................................ 48
  - Power: ** .................................................................................................. 48
  - Modulo: MOD ........................................................................................... 48
  - Complement: ~ ......................................................................................... 49
  - Shift Left: SHL and Shift Right: SHR ....................................................... 49
  - Bitwise AND: & ....................................................................................... 49
  - Bitwise OR: | .......................................................................................... 50
  - Bitwise XOR: ^ ....................................................................................... 50
  - Logical Conjunction: AND ....................................................................... 51
  - Logical Disjunction: OR ............................................................................ 51
  - Logical Negation: NOT ............................................................................. 52
  - Comparisons: <, >, <=, >=, ==, <> ......................................................... 52

**Setting the Evaluation Order of Operators** ................................................ 53

**Implicit Data Type Conversion** ................................................................... 53

**Examples for Expressions** .......................................................................... 54

**Using Conditional Program Branching Structures** .................................... 55

- **Simple Conditional Structure** ................................................................. 55
- **Nested Conditional Structure** ................................................................. 55
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changing Case of Strings</td>
<td>74</td>
</tr>
<tr>
<td>StringLower() Function</td>
<td>74</td>
</tr>
<tr>
<td>StringUpper() Function</td>
<td>74</td>
</tr>
<tr>
<td>Removing Spaces from Strings</td>
<td>74</td>
</tr>
<tr>
<td>StringTrim() Function</td>
<td>74</td>
</tr>
<tr>
<td>Formatting Strings with Spaces</td>
<td>75</td>
</tr>
<tr>
<td>Converting Between Characters and ASCII Codes</td>
<td>76</td>
</tr>
<tr>
<td>StringChar() Function</td>
<td>76</td>
</tr>
<tr>
<td>StringASCII() Function</td>
<td>76</td>
</tr>
<tr>
<td>Searching and Replacing Text in Strings</td>
<td>76</td>
</tr>
<tr>
<td>StringInString() Function</td>
<td>77</td>
</tr>
<tr>
<td>StringReplace() Function</td>
<td>77</td>
</tr>
<tr>
<td>Returning Information about Strings</td>
<td>78</td>
</tr>
<tr>
<td>StringLen() Function</td>
<td>79</td>
</tr>
<tr>
<td>StringTest() Function</td>
<td>79</td>
</tr>
<tr>
<td>Comparing Strings</td>
<td>79</td>
</tr>
<tr>
<td>StringCompare() Function</td>
<td>80</td>
</tr>
<tr>
<td>StringCompareNoCase() Function</td>
<td>80</td>
</tr>
<tr>
<td>StringCompareEncrypted() Function</td>
<td>81</td>
</tr>
<tr>
<td>Converting Data Types</td>
<td>81</td>
</tr>
<tr>
<td>Text() Function</td>
<td>81</td>
</tr>
<tr>
<td>StringFromIntg() Function</td>
<td>82</td>
</tr>
<tr>
<td>StringFromReal() Function</td>
<td>82</td>
</tr>
<tr>
<td>StringToIntg() Function</td>
<td>83</td>
</tr>
<tr>
<td>StringToReal() Function</td>
<td>84</td>
</tr>
<tr>
<td>DText() Function</td>
<td>84</td>
</tr>
<tr>
<td>Working with InTouch Windows at Run Time</td>
<td>85</td>
</tr>
<tr>
<td>Expose Window Name Property</td>
<td>85</td>
</tr>
<tr>
<td>GetWindowName() Function</td>
<td>85</td>
</tr>
<tr>
<td>Showing a List of Open Windows</td>
<td>86</td>
</tr>
<tr>
<td>OpenWindowList() Function</td>
<td>86</td>
</tr>
<tr>
<td>Checking If a Window is Open, Closed, or Exists</td>
<td>87</td>
</tr>
<tr>
<td>WindowState() Function</td>
<td>87</td>
</tr>
<tr>
<td>Opening InTouch Windows</td>
<td>87</td>
</tr>
<tr>
<td>Show() Function</td>
<td>87</td>
</tr>
<tr>
<td>ShowAt() Function</td>
<td>88</td>
</tr>
<tr>
<td>ShowHome() Function</td>
<td>88</td>
</tr>
<tr>
<td>ShowTopLeftAt() Function</td>
<td>88</td>
</tr>
<tr>
<td>Moving and Resizing a Window</td>
<td>89</td>
</tr>
<tr>
<td>WWWMoveWindow() Function</td>
<td>89</td>
</tr>
<tr>
<td>Hiding InTouch Windows</td>
<td>89</td>
</tr>
<tr>
<td>Hide() Function</td>
<td>90</td>
</tr>
<tr>
<td>HideSelf() Function</td>
<td>90</td>
</tr>
<tr>
<td>Changing the Color of a Window</td>
<td>90</td>
</tr>
<tr>
<td>ChangeWindowColor() Function</td>
<td>90</td>
</tr>
<tr>
<td>Printing Windows at Run Time</td>
<td>91</td>
</tr>
<tr>
<td>SetWindowPrinter() Function</td>
<td>91</td>
</tr>
<tr>
<td>Recommendations for Printing</td>
<td>91</td>
</tr>
<tr>
<td>PrintWindow() Function</td>
<td>91</td>
</tr>
<tr>
<td>PrintScreen() Function</td>
<td>92</td>
</tr>
<tr>
<td>PrintHT() Function</td>
<td>93</td>
</tr>
<tr>
<td>Starting Tag Viewer</td>
<td>93</td>
</tr>
</tbody>
</table>
### Contents

- **LaunchTagViewer() Function** .............................................................................................................. 93
- **Working with Date and Time Information** .............................................................................................. 94
  - Retrieving Numerical Date and Time Information ................................................................................. 94
    - $Year System Tag ................................................................................................................................. 94
    - $Month System Tag ............................................................................................................................... 95
    - $Day System Tag .................................................................................................................................... 95
    - $Hour System Tag ................................................................................................................................. 95
    - $Minute System Tag ............................................................................................................................. 96
    - $Second System Tag ............................................................................................................................... 96
    - $Msec System Tag .................................................................................................................................. 96
    - $Time System Tag ................................................................................................................................. 96
    - $Date System Tag .................................................................................................................................... 97
    - $DateTime System Tag ........................................................................................................................... 97
    - DateTimeGMT() Function ....................................................................................................................... 97
  - Retrieving String Date and Time Information ........................................................................................ 97
    - $DateString System Tag ......................................................................................................................... 98
    - $TimeString System Tag ......................................................................................................................... 98
    - UTCDatetime() Function ......................................................................................................................... 98
  - Converting Date and Time Information to Strings ................................................................................ 99
    - StringFromTime() Function .................................................................................................................... 99
    - wwsStringFromTime() Function ........................................................................................................... 100
    - StringFromTimeLocal() Function .......................................................................................................... 101
  - Checking the Daylight Savings Time Status .......................................................................................... 101
    - wwsIsDaylightSaving() Function ......................................................................................................... 101
- **Interacting with Other Applications** ................................................................................................... 102
  - Starting a Windows Application ............................................................................................................... 102
  - Retrieving the Application Title of a Running Application ................................................................ 102
    - InfoAppTitle() Function ....................................................................................................................... 102
  - Checking If an Application is Running .................................................................................................. 103
    - InfoAppActive() Function ...................................................................................................................... 103
  - Activating a Running Windows Application ......................................................................................... 103
    - ActivateApp Function ............................................................................................................................ 103
  - Sending Simulated Key Strokes to an Application ................................................................................ 104
    - SendKeys Function ................................................................................................................................. 104
  - Closing, Minimizing or Maximizing a Windows Application ................................................................ 105
    - WWControl() Function .......................................................................................................................... 105
  - Executing Commands and Exchanging Data using DDE ....................................................................... 106
    - WWExecute() Function .......................................................................................................................... 106
    - WWRequest() Function .......................................................................................................................... 107
    - WW Poke() Function .............................................................................................................................. 108
- **Working with Files** ............................................................................................................................... 108
  - Managing Files ......................................................................................................................................... 109
    - FileCopy() Function ............................................................................................................................... 109
    - FileDelete() Function ............................................................................................................................ 110
    - FileMove() Function ............................................................................................................................... 110
  - Reading and Writing CSV Data .............................................................................................................. 111
    - FileReadFields() Function .................................................................................................................... 111
    - FileWriteFields() Function .................................................................................................................... 112
  - Reading and Writing Text Data .............................................................................................................. 113
    - FileReadMessage() Function ................................................................................................................ 113
    - FileWriteMessage() Function .............................................................................................................. 114
  - Retrieving System-Related Information ................................................................................................. 114
Chapter 7 Scripting with OLE Objects ................................................. 127

About Scripting with OLE Objects .................................................. 127

Creating, Validating, and Releasing OLE Objects ............................ 127

OLE_CreateObject() Function ...................................................... 127
OLE_IsObjectValid() Function ...................................................... 128
OLE_ReleaseObject() Function ..................................................... 128

Using OLE Object Properties and Methods .................................. 129

Accessing the Properties of an OLE Object .................................. 129
Reading an OLE Object Property .................................................. 129
Writing to an OLE Object Property .............................................. 129

Calling Methods of an OLE Object .............................................. 130

Assigning Multiple Pointers to the Same OLE Object ...................... 130

Troubleshooting OLE Errors ....................................................... 130

OLE_GetLastErrorObject() Function ............................................ 131
OLE_GetLastErrorObjectErrorMessage() Function .......................... 131
OLE_ResetObjectError() Function ............................................... 131

OLE_ShowMessageOnObjectError() Function .................................. 131

OLE_IncrementOnObjectError() Function ...................................... 132

Things You Can Do with OLE ...................................................... 132

Produce Random Numbers ....................................................... 132
Create User Interface Dialog Boxes ................................................................. 132
Open Windows Date and Time Properties Panel ............................................... 134
Read and Write to the Registry ........................................................................ 134
Minimize Windows ......................................................................................... 135

Chapter 8 Scripting ActiveX Controls ............................................................... 137

About Scripting Active X Controls ................................................................. 137
Calling ActiveX Control Methods .................................................................... 137
Accessing ActiveX Control Properties from the InTouch HMI ......................... 138
    Configuring ActiveX Control Properties to Read and Write Data ................ 138
    Configuring Scripts to Read and Write ActiveX Control Properties .............. 138
    Linking ActiveX Control Properties to Tag or I/O References ....................... 139
Creating and Re-using ActiveX Event Scripts ............................................... 140
    Creating ActiveX Event Scripts .................................................................. 140
    Re-using ActiveX Event Scripts .................................................................. 141
    Creating Self-Referencing ActiveX Event Scripts ........................................ 142
Importing ActiveX Event Scripts .................................................................... 143

Chapter 9 Troubleshooting QuickScripts ......................................................... 145

Logging Messages to the Log Viewer ............................................................... 145
    LogMessage() Function ........................................................................... 145
Viewing Log Viewer Messages ......................................................................... 146

Index .................................................................................................................. 148
InTouch HMI Scripting and Logic Guide

CHAPTER 1

Introduction to Scripting

In This Chapter

About Scripting ........................................................................................................................................ 11
Basic Scripting Concepts .......................................................................................................................... 11
Advanced Scripting Concepts .................................................................................................................... 12

About Scripting

You can use the InTouch scripting language, QuickScript, to build more robust applications. There are eight types of scripts and many built-in script functions available.

The eight types of scripts are defined by what causes them to execute. For example, application scripts execute when an application starts, stops, or continues running. Data change scripts execute when a certain item of data changes. Window scripts execute when a window opens, closes, or remains open.

The built-in script functions include mathematical functions, trigonometric functions, string functions, and others. Using these functions saves you time in developing your application.

InTouch scripts can include Object Linking and Embedding (OLE) objects and ActiveX controls.

You can use conditional statements, loops, and local variables in the scripting language to create complex effects in your application.

Basic Scripting Concepts

Before you start scripting, you should understand:

• A script is a set of instructions that direct an application to do something.
• QuickScript is the InTouch HMI scripting language.
• A function is a script that can be called by another script. The InTouch HMI comes with a set of predefined functions for your use.
• QuickFunctions are re-usable functions written in QuickScript and stored in the QuickFunction library. To create a QuickFunction, you simply create a QuickScript and name it. A QuickFunction can be called by another script or from animation link expressions.

Types of Scripts

In InTouch, scripts are categorized based on what causes the script to run. For example, you would create a "key script" if you want a script to execute when the operator presses a certain key on the keyboard.

After you have chosen the script type, you can then further define the criteria, or conditions, that make the script execute. For example, you might want the script to execute when the key is released, not when the key is pressed.

The script types are:

• Application scripts execute either continuously while WindowViewer is running or one time when WindowViewer is started or shut down.
- **Window** scripts execute periodically when an InTouch window is open or one time when an InTouch window is opened or closed.
- **Key** scripts execute one time or periodically when a certain key or key combination is pressed or released.
- **Condition** scripts execute one time or periodically when a certain condition is fulfilled or not fulfilled.
- **Data change** scripts execute one time when the value of a certain tag or expression changes.
- **Action** scripts execute one time or periodically when an operator clicks on an InTouch HMI graphic object. Action scripts often are used for pushbuttons.
- **ActiveX event** scripts execute one time when an ActiveX event occurs, such as clicking the ActiveX control.

**Advanced Scripting Concepts**

Some advanced scripting capabilities allow you to achieve sophisticated functions beyond those of the basic InTouch HMI.

OLE objects and ActiveX controls allow you to access your native computer system functions and interact with other programs such as the Manufacturing Engineering Module.

**OLE Objects**

In your custom scripts, you can call OLE objects. OLE objects allow you to access your native computer system functions and to interact with other programs such as the Wonderware Manufacturing Engineering Module.

For example, using OLE, you can:
- Produce random numbers.
- Create user interface dialog boxes.
- Open the Windows date and time properties panel.
- Read and write to the registry.
- Minimize windows.

**Scripting with ActiveX Controls**

Several ActiveX controls are provided with the InTouch HMI in the Wizards menu. Because the InTouch HMI is based on the Windows operating environment, you can use nearly any ActiveX control with the InTouch HMI.
In Touch HMI Scripting and Logic Guide

CHAPTER 2

Creating and Editing Scripts

In This Chapter

About Creating and Editing Scripts ................................................................. 13
Working with the InTouch Script Editor ............................................................ 14
Opening a Script for Editing ........................................................................... 17
Saving or Discarding Changes to a Script ....................................................... 18
Copying, Cutting and Pasting Text ................................................................. 19
Searching Within a Script ............................................................................... 19
Inserting Code Elements ............................................................................... 26
Accessing Help for Script Functions ............................................................. 27
Validating Scripts for Correct Syntax ............................................................ 27
Printing Scripts ............................................................................................. 27
Deleting Scripts ............................................................................................. 28

About Creating and Editing Scripts

The steps to create a new script vary according to the script type. In general, you open the Script Editor, select a condition type, enter statements, and then save the script.

For detailed information on creating scripts of each type, see the following sections:

- Configuring Application Scripts on page 30.
- Configuring Window Scripts on page 32.
- Configuring Key Scripts on page 33.
- Configuring Condition Scripts on page 34.
- Configuring Data Change Scripts on page 36.
- Configuring Action Scripts on page 36.
- Configuring ActiveX Event Scripts on page 39.
## Working with the InTouch Script Editor

Use the InTouch HMI Script Editor to create and edit scripts within InTouch WindowMaker.

![InTouch Script Editor](image)

<table>
<thead>
<tr>
<th>Area</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Toolbars</td>
</tr>
</tbody>
</table>
| 2    | Condition definition area  
The **Condition Type** box provides the available execution conditions for the type of script you are writing. |
| 3    | Script text window         |
| 4    | Command buttons            |
| 5    | Built-in script function buttons |
| 6    | MEM OLE button  
The **MEM OLE** button in the lower right corner only appears if the Manufacturing Engineering Module (MEM) is installed with the InTouch HMI installation. Clicking this button allows you to script with MEM. |

This example is for an application script. Each type of script has its own version of the script dialog box, with options and selections that are unique to that type of script.

The title bar of the editor identifies which type of script you are working with. For information about types of scripts, see *Types of Scripts* on page 11.
There are text, equivalency and mathematical operator buttons available in a context-sensitive pop-up window that you can click to insert that keyword, function, or symbol into your script at the cursor location.

**Color Indicators for InTouch Script Elements**

The InTouch script 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></td>
<td>Syntax highlighted while typing.</td>
</tr>
<tr>
<td>Comments (both single line and multi-line)</td>
<td>Green</td>
</tr>
<tr>
<td></td>
<td>Syntax highlighted while typing.</td>
</tr>
<tr>
<td>Strings</td>
<td>Purple</td>
</tr>
<tr>
<td></td>
<td>Syntax highlighted while typing.</td>
</tr>
<tr>
<td>Function names, numeric constants, operators, semicolons, dim variables, alias variables, and so on</td>
<td>Black</td>
</tr>
<tr>
<td></td>
<td>See descriptions for Attribute names and Reserved words.</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>

**InTouch Script Editor Autocomplete Features**

The InTouch Script Editor autocomplete incorporates several features for use while authoring InTouch scripts:

- Provides an autocomplete tag reference in a selectable list box.
- Provides method parameter help in an autocomplete list box including context-specific suggestions covering definitions, keywords, script elements, and programmatic constructs.
- Automatic word completion of tags, dotfields, 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 at the bottom of the autocomplete pop-up window.
Accepting InTouch Script Editor 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 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 in InTouch Scripting

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

- Use main menu options Edit, then Undo or Ctrl+Z to undo edits. You can also use context menu options to undo and redo.
- Use main menu options Edit, then Redo or Ctrl+Y to redo edits. You can also use context menu options to undo and redo.

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 are lost if you close the Script Editor or switch to another script within the Script Editor.
Visual Indication of InTouch Script Errors

Errors in InTouch script text are marked with a red "squiggly" underline.

Hovering over the error with the mouse cursor will display the error message as a tooltip. The tooltip error message provides the same information as the message shown when clicking the script verification button.

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.

When the "\" character is used to concatenate two strings to form a path, the script editor displays a red underline under "\". This is an exception and can be ignored. The path is constructed correctly in runtime.

Managing InTouch Script Edits

The InTouch Script Editor offers tools and visual references to find and manage edits.

Script Line Numbers

The Script Editor displays line numbers at the left margin.

- Line numbers up to four digits appear when the Script Editor is not zoomed.
- Use the right-click context menu Go To option to go to a specific line in a script.
- Delete the current line of text by pressing Ctrl+L.
- You can select and drag a line of text to a different line.

Change Bars

As a visual reference for script changes in progress, yellow change bars on the left margin of the script text window indicate additions and line insertions and edits.

Tag Definition

You can enter a new tagname in the InTouch Script Editor, then press Ctrl+T or click Edit on the main menu to define a tag. The Tagname Dictionary appears so you can complete the tagname definition.

Find and Replace

The InTouch Script Editor provides customizable find and replace functionality. For information on how to use the find and replace feature, see Searching Within a Script on page 19.

Opening a Script for Editing

The steps to open an existing script vary slightly depending on the script type.

To open an application script

1. Do either of the following:
   - Using the Classic View, in the Scripts pane, double-click Application.
   - On the Special menu, point to Scripts, and then click Application Scripts.
2. In the Condition Type list, click the type of script to edit.

To open a window script

1. Do any of the following:
Creating and Editing Scripts

InTouch HMI Scripting and Logic Guide

1. To open a script:
   o Using the Classic View, in the Windows pane, right-click the window name, and then click Window Scripts.
   o Using the Project View, expand Scripts, and then double-click the script.
   o Open the window that the script is associated with. On the Special menu, point to Scripts, and then click Window Scripts.
   o Open the window that the script is associated with. Right-click on a blank area in the window, and then click Window Scripts.

2. In the Condition Type list, click the condition to cause the script to run.

To open an ActiveX event script

- Do any of the following:
  o Using the Classic View, in the Scripts pane, expand ActiveX Event, and then double-click the script name.
  o Using the Project View, expand Scripts, and then double-click the script.
  o Double-click the ActiveX control instance that the script is associated with. Click the Events tab, and then double-click the cell that contains the script name.

To open an action script

1. Open the window that contains the graphic element that the action script is associated with. A typical use for an action script would be to script an action on a button.
2. Double-click the graphic element that the action script is associated with.
3. In the Touch Pushbuttons area, click Action. The Script Editor appears.
4. In the Condition Type list, click the action to cause the script to run.

To open key, condition, or data change scripts

1. Do any of the following:
   o Using the Classic View, in the Scripts pane, expand the script category, and then double-click the script name.
   o Using the Project View, expand Scripts, and then double-click the script.
   o On the Special menu, point to Scripts, and then click the relevant script type. The Script Editor appears. Click the Browse button, and click the script name.
2. If applicable, in the Condition Type list, click the condition to cause the script to run.

Saving or Discarding Changes to a Script

While working in the Script Editor, or when finished, you can save your script either manually or automatically. Or you can discard it altogether.

The restore option is not available for window and application scripts.

Note: Saving and discarding changes always applies to all condition types for a type of script, not just the condition type that is currently visible.

To save changes and keep the script open

- On the Script menu, click Save.

To save changes and close the script

- Click OK.
To discard changes and keep the script open
- Click Restore.

To discard changes and close the script
- Click Cancel.

Copying, Cutting and Pasting Text

Copying, cutting and pasting text in the Script Editor works the same way as in any other Windows application. Use either the standard keyboard shortcuts Ctrl-C, Ctrl-X, and Ctrl-V, or the toolbar buttons. You can also select a line and drag the text in that line to a new position in the script.

Searching Within a Script

The InTouch Script Editor provides the functionality to find and replace text within an open script based on a search string and a set of customizable search options. You can access Find and Replace as Edit options on the Script Editor’s menu bar.

You can also use keyboard shortcuts Ctrl+F (Quick find) and Ctrl+H (Quick replace) to display the Find and Replace dialog box.
- In a simple search using default search options, select the Quick Find tab and type a word or phrase in the Find what field.
Select **Find Next** to start a search. A light blue background identifies a word or phrase in the script that matches the search string.

```
121 IF Cycle -- 3 THEN
122 HorizontalMove = HorizontalMove + Speed * 3;
123 ENDIF;
124
125 IF Auto THEN
126 IF HorizontalMove > 275 THEN
127 Ejector = 1;
128 IF HorizontalMove > 239 THEN
129 HorizontalMove = 239;
130 Cycle = 4;
131 ENDIF;
132
133 ELSE (AUTO)
134 IF Ejector THEN
135 IF (HorizontalMove > 250) AND
136 Cycle = 7;
137 ENDIF;
138
139 IF (HorizontalMove > 272) AND
140 HorizontalMove = 239;
141 Cycle = 4;
142 ENDIF;
143
144 ELSE
145 IF HorizontalMove > 320 THEN
146 Cycle = 6;
```

- In a simple replace operation using default options, select the **Quick Replace** tab, type a word or phrase in the **Find what** field, and also type a replacement word or phrase in the **Replace with** field.

```
15 IF Auto THEN
16 IF Step1 = 11 THEN
17 Step1 = 1;
18 ENDIF;
19
20 IF Step1 = 1 THEN
21 InputValve = 1;
22 ConcPump = 0;
23 ConcValve = 0;
24 SteamValve = 0;
25 Mixer = 0;
26 TransferPump = 0;
27 TransferValve = 0;
28 OutputValve = 0;
29 Step2 = 2;
30 ENDIF;
31
32 IF Step1 = 2 THEN
33 IF ReactLevel >= (2000 - (P
34 Step1 = 3;
35 ENDIF;
36
37 IF Step1 = 3 THEN
```

Select **Find Next** to start a replacement operation. A light blue background identifies a word or phrase in the script that matches the search string entered in the **Find what** field. You have three replacement options after locating a matching word or phrase.

- **Select Find Next** to ignore the current matching word or phrase and continue searching for the next match in the script.
Select **Replace** to replace the current matching word or phrase with the string entered in the **Replace with** field.

Select **Replace All** to replace all matching words or phrases with the string entered in the **Replace with** field.

### Configuring a Find or Replace

The **Find and Replace** dialog box provides a set of options to configure a set of search options.

#### Look in

The **Look in** field includes options to search the entire script (**Document**) or only a selected portion of a script (**Selection**). **Document** is the default.

When you want to search only within a selected portion of a script, select the search area within the script using your mouse before conducting the search. A blue background identifies the selected script lines, which can be searched. The search results only show matching items within the portion of the script you selected.

#### Find Options

You can expand or collapse the **Find options** section. The following options can be selected or cleared to filter search results more precisely.

- **Match case**
  
  When selected, the search results only display instances of the **Find what** string that are matched both by content and by case. For example, a search for Triangle4 with **Match case** selected returns Triangle4 but not triangle4.

- **Match whole word**

  When selected, the search results only display instances of the **Find what** string that are matched in complete words. For example, a search for LogicBit will return LogicBit but not LogicBits.

- **Search up**

  When selected, a search is conducted from the current position within a script to the top of the script. By default, a search is conducted from the current position within a script to the bottom.

#### Search type

The **Search type** field provides options to conduct a script search based on the type of search.

- **Normal**

  The default search type, which requires an exact match between the characters in a search string and text in a script.

- **Search by Regular Expressions** on page 22

  A regular expression describes one or more strings to match when you search a script. A regular expression consists of ordinary characters that serve as a template for matching a character pattern to the string being searched.

- **Search by Wildcard Characters** on page 24
A wildcard search uses keyboard characters like an asterisk (*) or a question mark (?) to represent one or more characters when searching within a script.

- **Search by Acronym** on page 25
  An acronym search matches a character at the start of a word, then every capital letter or character following an underscore.

- **Search by Shorthand** on page 26
  A shorthand search extends the 'Search by Acronym' option by allowing non-whitespace characters between the search pattern characters.

You must type a search string that complies with the syntax and supported characters of the selected search type.

**Search by Regular Expressions**

A regular expression describes one or more strings to match when you search a script using alphanumeric characters and special characters known as metacharacters. The regular expression serves as character pattern to compare with the script text being searched.

Regular expressions are constructed much like arithmetic expressions are created. Small expressions are combined by using a variety of metacharacters and operators to create larger expressions.

The components of a regular expression can be individual characters, sets of characters, ranges of characters, or choices between characters. Components can also be any combination of these components.

**Script Regular Expressions**

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Match any single character (except a line break)</td>
<td>s.e matches &quot;ste&quot; in &quot;step&quot; and &quot;sfe&quot; in &quot;transfer&quot; but not &quot;acro&quot; in &quot;across&quot;.</td>
</tr>
<tr>
<td>*</td>
<td>Match zero or more occurrences of the preceding expression (match as many characters as possible)</td>
<td>a*r matches &quot;r&quot; in &quot;rack&quot;, &quot;ar&quot; in &quot;ark&quot;, and &quot;aar&quot; in &quot;aardvark&quot;</td>
</tr>
<tr>
<td>*</td>
<td>Match any character zero or more times (Wildcard *)</td>
<td>c.*e matches &quot;cke&quot; in &quot;racket&quot;, &quot;comme&quot; in &quot;comment&quot;, and &quot;code&quot; in &quot;code&quot;</td>
</tr>
<tr>
<td>+</td>
<td>Match one or more occurrences of the preceding expression (match as many characters as possible)</td>
<td>e.+e matches &quot;eede&quot; in &quot;feeder&quot; but not &quot;ee&quot;.</td>
</tr>
<tr>
<td>.+</td>
<td>Match any character one or more times (Wildcard ?)</td>
<td>e.+e matches &quot;eede&quot; in &quot;feeder&quot; but not &quot;ee&quot;.</td>
</tr>
<tr>
<td>*?</td>
<td>Match zero or more occurrences of the preceding expression (match as few characters as possible)</td>
<td>e.*?e matches &quot;ee&quot; in &quot;feeder&quot; but not &quot;eede&quot;.</td>
</tr>
<tr>
<td>+?</td>
<td>Match one or more occurrences of the preceding expression (match as few characters as possible)</td>
<td>e.+?e matches &quot;ente&quot; and &quot;erprise&quot; in &quot;enterprise&quot;, but not the whole word &quot;enterprise&quot;.</td>
</tr>
<tr>
<td>Character</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>^</td>
<td>Anchor the match string to the beginning of a line or string</td>
<td><code>^car</code> matches the word &quot;car&quot; only when it appears at the beginning of a line.</td>
</tr>
<tr>
<td>\r?$</td>
<td>Anchor the match string to the end of a line</td>
<td>End</td>
</tr>
<tr>
<td>[abc]</td>
<td>Match any single character in a set</td>
<td>b[abc] matches &quot;ba&quot;, &quot;bb&quot;, and &quot;bc&quot;.</td>
</tr>
<tr>
<td>[a-t]</td>
<td>Match any character in a range of characters</td>
<td>be[n-t] matches &quot;bet&quot; in &quot;between&quot;, &quot;ben&quot; in &quot;beneath&quot;, and &quot;bes&quot; in &quot;beside&quot;, but not &quot;below&quot;.</td>
</tr>
<tr>
<td>(</td>
<td>Capture and implicitly number the expression contained within parenthesis</td>
<td>([a-z])X.1 matches &quot;aXa&quot;and &quot;bXb&quot;, but not &quot;aXb&quot;. &quot;1&quot; refers to the first expression group [a-z]].</td>
</tr>
<tr>
<td>(?abc)</td>
<td>Invalidate a match</td>
<td>real (?lity) matches &quot;real&quot; in &quot;reality&quot; and &quot;really&quot; but not in &quot;reality.&quot; It also finds the second &quot;real&quot; (but not the first &quot;real&quot;) in &quot;realityreal&quot;.</td>
</tr>
<tr>
<td>[^abc]</td>
<td>Match any character that is not in a given set of characters</td>
<td>be[\n-t] matches &quot;bef&quot; in &quot;before&quot;, &quot;beh&quot; in &quot;behind&quot;, and &quot;bel&quot; in &quot;below&quot;, but not &quot;beneath&quot;.</td>
</tr>
<tr>
<td></td>
<td>Match either the expression before or the one after the symbol.</td>
<td>(sponge&amp;</td>
</tr>
<tr>
<td>^</td>
<td>Escape the character following the backslash</td>
<td></td>
</tr>
<tr>
<td>{x}</td>
<td>Specify the number of occurrences of the preceding character or group</td>
<td>x(ab){2} matches &quot;xababx&quot;, and x(ab){2,3} matches &quot;xabababx&quot; and &quot;xabababxb&quot; but not &quot;xababababx&quot;.</td>
</tr>
<tr>
<td>\p{X}</td>
<td>Match text in a Unicode character class, where &quot;X&quot; is the Unicode number.</td>
<td>\p{Lu} matches &quot;T&quot; and &quot;D&quot; in &quot;Thomas Doe&quot;.</td>
</tr>
<tr>
<td>\b</td>
<td>Match a word boundary</td>
<td>\bin matches &quot;in&quot; in &quot;inside&quot; but not &quot;pinto&quot;.</td>
</tr>
<tr>
<td>\r?\n</td>
<td>Match a line break (ie a carriage return followed by a new line).</td>
<td>End</td>
</tr>
<tr>
<td>\w</td>
<td>Match any alphanumerical character</td>
<td>a\wd matches &quot;add&quot; and &quot;a1d&quot; but not &quot;a d&quot;.</td>
</tr>
<tr>
<td>(?[^\r\n])*s</td>
<td>Match any whitespace character.</td>
<td>Public\s\sInterface matches the phrase &quot;Public Interface&quot;.</td>
</tr>
<tr>
<td>\d</td>
<td>Match any numeric character</td>
<td>\d matches and &quot;3&quot; in &quot;3456&quot;, &quot;2&quot; in 23&quot;, and &quot;1&quot; in &quot;1&quot;.</td>
</tr>
</tbody>
</table>
Creating and Editing Scripts

InTouch HMI Scripting and Logic Guide

\uXXXX where XXXX specifies the Unicode character value

| Match a Unicode character | \u0065Â matches the character “e” |

\b\(w+\|w-[0-9]\]\w*)\b

| Match an identifier | Matches “type1” but not &type1” or “#define”. |

\b([^\s]+?\s\(\|\s)?((\|\s)+))

| Match a string inside quotes | Matches any string inside single or double quotes. |

\b0[xX][[0-9a-fA-F]]\b

| Match a hexadecimal number | Matches ”0xc67f” but not ”0xc67fc67f” |

\b[0-9]+\\[0-9]+\b

| Match integers and d | Matches ”1.333” |

Order of Precedence

A regular expression is evaluated from left to right and follows an order of precedence.

The following table contains the order of precedence of regular expression operators, from highest to lowest.

<table>
<thead>
<tr>
<th>Operator or operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Escape</td>
</tr>
<tr>
<td>(), (?), (?=), []</td>
<td>Parentheses and brackets</td>
</tr>
<tr>
<td>*, +, ?, {n}, {n}, {n,m}</td>
<td>Quantifiers</td>
</tr>
<tr>
<td>^, $, \anymetacharacter</td>
<td>Anchors and sequences</td>
</tr>
<tr>
<td></td>
<td>Alternation</td>
</tr>
</tbody>
</table>

Characters have higher precedence than the alternation operator, which, for example, allows "m|food" to match "m" or "food".

Search by Wildcard Characters

A wildcard search uses a single keyboard character, which can be interpreted as a number of literal characters or an empty string when you are searching for a string in a script.

Wildcard characters are often used in place of one or more characters when you do not know what the real character is or you do not want to type the entire search string.

<table>
<thead>
<tr>
<th>Wildcard character</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asterisk (*)</td>
<td>An asterisk in a search string matches any sequence of characters. Use the asterisk as a substitute for zero or more characters.</td>
</tr>
</tbody>
</table>
| | \begin{itemize}
| | • Logic*
| | • *Test*
| | \end{itemize} |
| Question mark (?) | A question mark in a search string matches any character at a
Creating and Editing Scripts

InTouch HMI Scripting and Logic Guide

Search by Acronym

Acronym searching matches a character at the start of a word, then every capital letter or character following an underscore.

```
IF NOT LogicBits.11 THEN
{Simulate Triangle Waves}
IF LogicBits.01 THEN
    IF Triangle1 > 0 THEN
        Triangle1 = Triangle1 - 1;
    ELSE
        LogicBits.01 = 0;
    ENDIF;
ELSE
    IF Triangle1 < 100 THEN
        Triangle1 = Triangle1 + 1;
    ELSE
        LogicBits.01 = 1;
    ENDIF;
ENDIF;
ELSE
    IF Triangle2 > 0 THEN
        Triangle2 = Triangle2 - 1;
    ELSE
        LogicBits.02 = 0;
    ENDIF;
ELSE
    IF Triangle2 < 100 THEN
        Triangle2 = Triangle2 + 1;
    ELSE
        LogicBits.02 = 1;
    ENDIF;
ENDIF;
```

Examples: The search term "LB" finds instances of "LogicBits".
Search by Shorthand

The shorthand search extends the ‘Search by Acronym’ option, by allowing any number of non-whitespace characters between the search pattern characters.

```
1 IF NOT LogicBits.11 THEN
2 {Simulate Triangle Waves}
3 IF LogicBits.01 THEN
4   IF Triangle1 > 0 THEN
5     Triangle1 = Triangle1 - 1;
6   ELSE
7     LogicBits.01 = 0;
8 ENDIF;
9 ELSE
10   IF Triangle1 < 100 THEN
11     Triangle1 = Triangle1 + 1;
12   ELSE
13     LogicBits.01 = 1;
14 ENDIF;
15 ENDIF;
16 IF LogicBits.02 THEN
17   IF Triangle2 > 0 THEN
18     Triangle2 = Triangle2 - 1;
19   ELSE
20     LogicBits.02 = 0;
21 ENDIF;
22 ELSE
23   IF Triangle2 < 100 THEN
24     Triangle2 = Triangle2 + 1;
25 ENDIF;
26 ENDIF;
```

Examples: The search term “ta” finds instances of “Triangle”.

Inserting Code Elements

You can automatically insert various code elements into your script by selecting them from lists. This saves you time and reduces the risk of typing errors. Access the code elements one of two ways: using the main menu Insert options, or by using the autocomplete pop-up windows and selecting your element from the list.

Code elements you can insert, both from autocomplete pop-up windows and from the Script Editor menu items include:

- Functions
- Tagnames and Dotfields For a dotfield, type a tagname followed by a period to open an autocomplete pop-up window.
- Variable
- Window name
- ActiveX instances
- Keyword
- Operator

The autocomplete pop-up windows include a row of icons at the bottom of the pop-up for direct access to the available elements.
Accessing Help for Script Functions

If you are looking for help on a specific script function, you can access it directly from the Script Editor.

**To view help on a specific script function**

1. In the bottom right corner of the Script Editor, click Help.
   
   A list of functions appears.

2. Click the name of the function from the list to show its help.
   
   The corresponding Help topic appears.

Validating Scripts for Correct Syntax

When you save a script, the Script Editor automatically checks it for correct syntax. If an error is discovered, a message with more information appears. You must fix all syntax errors before you can save the script. You can also start the validation manually while you are editing the script.

**To manually validate script syntax**

- Click Validate.

Highlighting Script Errors

The InTouch Script Editor also highlights script errors. For more information, see *Visual Indication of InTouch Script Errors*.

Printing Scripts

You can print scripts individually from the Script Editor, or you can print all scripts of a specific type using the print feature in WindowMaker.

You can print scripts individually from the Script Editor, or you can print all scripts of a specific type using the print feature in WindowMaker.

**To print an individual script**

1. Open the script in the Script Editor.
   
2. Click Print in the toolbar. The script is printed to the Windows default printer.

**To print all scripts of a specific type**

   
2. To print window scripts, do the following:
   
   a. Select Windows.

   b. Select the windows to print:

      All prints the information for all windows in the application.

      Selected prints only the information for specific windows. The Windows to Print dialog box appears. Select the windows in your application you want to print and click OK.

      Batch prints only the information for windows specified in a .csv file.

   c. Select Window Scripts to print the scripts associated with the windows.

3. To print other types of scripts, select the appropriate check boxes. To print all scripts, click All Scripts.

4. Click Next. The Select Output Destination dialog box appears.

5. Do one of the following
Creating and Editing Scripts

InTouch HMI Scripting and Logic Guide

- Click **Send output to Printer**.
- Click **Send output to Text File**.

6. Click **Browse** to select a printer or to find a file.

7. Click **Print**.

**To print all scripts**

1. Select **All Scripts** to print all scripts used in the application.

   You can restrict printing to only selected types of scripts by clearing the **All Scripts** check box. Then, select the check box for each type of script that you want to print.

2. Click **Next**. The **Select Output Destination** dialog box appears.

3. Select the option to print the contents of the Tagname Dictionary or send the output to a text or .html file.

4. Click **Print**.

**Deleting Scripts**

The steps to delete a script vary depending on the script type. See the following sections:

- **Configuring Application Scripts** on page 30.
- **Configuring Window Scripts** on page 32.
- **Configuring Key Scripts** on page 33.
- **Configuring Condition Scripts** on page 34.
- **Configuring Data Change Scripts** on page 36.
- **Configuring Action Scripts** on page 36.
- **Configuring ActiveX Event Scripts** on page 39.
About Script Triggers

All InTouch HMI scripts are executed by script triggers. Each script type has one or more triggers to launch it.

In the Script Editor, you can select which script trigger you want to use to execute your script. You select a script trigger based on when and how a script is executed.

You can configure various triggers based on user actions, internal states, and changes of tagname values. User actions include pressing keys and clicking on graphic elements. Internal state triggers can include starting WindowViewer.

Scripts are triggered by these kinds of actions:

- Starting and shutting down WindowViewer. See Configuring Application Scripts on page 30.
- Opening and closing a window. See Configuring Window Scripts on page 32.
- Pressing a key or key combination. See Configuring Key Scripts on page 33.
- Fulfilling a certain condition, such as tagname or an expression value. See Configuring Condition Scripts on page 34.
- Changing tagname values or tagname field values. See Configuring Data Change Scripts on page 36.
- Clicking a graphic object. See Configuring Action Scripts on page 36.
- Events that occur in an ActiveX control, such as clicking on the control. See Configuring ActiveX Event Scripts on page 39.

Also, you can pause script execution. By default, when WindowViewer is started, logic is running and scripts are executed. You can pause script execution at run time by halting logic. After pausing you can resume script execution. For more information, see Pausing Script Execution at Run Time on page 41.
Types of Script Triggers

In the InTouch HMI, scripts are divided into seven types. Each type of script has one or more triggers you can select to launch a script.

- An application script has three triggers: on startup, on shutdown, and while running. Each trigger can execute a different script.
- A window script has three triggers: on show, on hide, and while showing. Each trigger can execute a different script.
- A key script has three triggers: on key up, on key down, or while down. Each trigger can execute a different script.
- A condition script has four triggers: on true, while true, on false, and while false. Each trigger can execute a different script.
- A data change script executes when the value of a certain tag or expression changes.
- An action script executes one time or periodically when an operator clicks on an InTouch HMI graphic object.
- An ActiveX event script executes one time when a certain ActiveX event occurs, such as a click on the ActiveX control.

Using Multiple Triggers

For most script types you can use multiple triggers and associate different scripts with each trigger. For example, you can configure an application script to execute one script when WindowViewer is started, and another script periodically while WindowViewer is running.

Select the trigger in the Condition Type list to view the existing script for a trigger.

Periodic Script Execution

Scripts that execute periodically do not execute immediately after triggering, but after the specified period for the first time.

For example, if you configure a key script to execute every 5000 ms while a specific key is pressed, it executes 5 seconds after the key is pressed and held down and then every 5 seconds afterwards.

Configuring Application Scripts

Application scripts are linked to the entire InTouch HMI application. You can use application scripts to:

- Execute a script one time when WindowViewer is started.
- Execute a script periodically while WindowViewer is running.
- Execute a script one time when WindowViewer is shut down.
To configure an application script

1. Using the Classic View, in the Scripts pane, right-click on Application and then click Open. The Application Script dialog box appears.

2. In the Condition Type list, click the condition for the script execution:
   - Click On Startup to configure a script to execute one time when WindowViewer is started.
   - Click While Running to configure a script to execute periodically while WindowViewer is running.
   - Click On Shutdown to configure a script to execute one time when WindowViewer is shut down.
3. If you selected While Running in the previous step, type a time interval between 1 and 360000 milliseconds in the Every box. The time interval specifies how often the script is executed.
4. Type your script in the window.
5. Click OK.

To delete an application script

1. Using the Classic View, in the Scripts pane, right-click on Application and then click Open. The Application Script dialog box appears.
2. In the Condition Type list, click the condition for the script to delete. The script appears in the main section of the Application Script dialog box.
3. On the Edit menu, click Clear. The script from the main section clears and the associated script is deleted.

Limitations of Application Scripts

Application scripts that are executed when WindowViewer starts or shuts down have limitations on their interaction with other objects.

You cannot use On Startup application scripts to:
- Reference ActiveX methods, properties, or events.
- Read from or write to controls and I/O tag names or remote references.
- Run data change scripts and condition scripts.
  You cannot use On Shutdown application scripts to:
- Read from or write to controls and I/O tag names or remote references.
- Start other applications.

Configuring Window Scripts

Window scripts are scripts that are linked to specific windows. You can use the `GetWindowName` script function to help the run-time environment reduce scripting necessary to load windows. You can use window scripts to:
- Execute a script one time when an InTouch window is opened.
- Execute a script periodically while an InTouch window is open.
- Execute a script one time when an InTouch window is closed.

**Note:** Opening an InTouch window is also referred to as "showing an InTouch window." Closing an InTouch window is also referred to as "hiding an InTouch window."

**To configure a window script**

1. Using the Classic View, in the Windows pane, right-click on a window and then click Window Scripts. The Window Script for Window Name dialog box appears.

2. In the Condition Type list, do one of the following:
   - Click On Show to configure a script to execute one time when the associated window is started.
   - Click While Showing to configure a script to execute periodically while the associated Window is open.
   - Click On Hide to configure a script to execute one time when the associated window is closed.

3. If you select While Showing in the previous step, type a time interval between 1 and 360000 milliseconds in the Every box.

4. Type your script in the window.

5. Click OK.
To delete a window script

1. Using the Classic View, in the Windows pane, right-click on a window and click Window Scripts. The Window Script for Window Name dialog box appears.
2. In the Condition Type list, click the script trigger for the script to delete. The script appears in the main section of the Window Script for Window Name dialog box.
3. On the Edit menu, click Clear.

**Important:** Do not use on hide scripts to read from or write to I/O tag names. The I/O value update does not necessarily complete before the window is hidden.

To read from or write to I/O tag names when a window closes, configure a data change script and activate it from an on hide script.

---

Configuring Key Scripts

Key scripts are scripts that are linked to the pressing of a specific key or key combination. You can use key scripts to:

- Execute a script one time when a key or key combination is pressed.
- Execute a script periodically while a key or key combination is pressed and not released.
- Execute a script one time when a key or key combination is released.

A key script is identified by the name of key that initiates the script. For example: Ctrl+q.

**Note:** If you have configured an action script that uses the same key or key combination to trigger it, the key script is ignored and instead the action script is executed.

To configure a key script

1. Using the Classic View, in the Scripts pane, do one of the following:
   - To configure a new key script, right-click Key, and then click New. The Key Scripts dialog box appears.
   - To configure an existing key script, expand Key, right-click the script name, and then click Edit. The Edit Key Script dialog box appears.
2. Click **Key** and select a key from the **Choose Key** dialog box.

3. Select the **Ctrl** and/or **Shift** check boxes to assign a control key and/or shift key combination with your selected key.

4. In the **Condition Type** list, do one of the following:
   - Click **On Key Down** to configure a script to execute one time when the associated key or key combination is pressed.
   - Click **While Down** to configure a script to execute periodically while the associated key or key combination is pressed.
   - Click **On Key Up** to configure a script to execute one time when the associated key or key combination is released.

5. If you selected **While Down** in the previous step, type a time interval between 1 and 360000 milliseconds in the **Every** box.

6. Type your script in the window.

7. Click **OK**.

**To delete all key scripts associated with a key**

- Using the **Classic View**, in the **Scripts** pane, expand **Key**, right-click the key script name, and then click **Delete**. When a message appears, click **Yes**.

**To delete a key script that is associated with a key**

1. Using the **Classic View**, in the **Scripts** pane, expand **Key**, right-click the key script name, and then click **Edit**. The **Edit Key Script** dialog box appears.

2. In the **Condition Type** list, click the script trigger for the script to delete. The script appears in the main section of the **Edit Key Script** dialog box.

3. On the **Edit** menu, click **Clear**. The script from the main section clears and the associated script is deleted.

### Configuring Condition Scripts

Condition scripts are triggered depending on when certain logical conditions are fulfilled. Use condition scripts to execute a script:

- One time when a condition is fulfilled.
- One time when a condition is not fulfilled.
- Periodically while a certain condition is fulfilled.
- Periodically while a certain condition is not fulfilled.

A condition script is identified by the condition syntax that initiates the script. For example: tag1>=13.

**Note**: A script that is assigned the **On True** condition type only executes if the condition transitions from False to True. A script that is assigned the **On False** condition type only executes if the condition transitions from True to False.
To configure a condition script

1. Using the **Classic View**, in the **Scripts** pane either:
   - Right-click **Condition** and click **New**. The **Condition Scripts** dialog box appears.
   - To edit an existing condition script, click the plus sign next to **Condition**, right-click the condition script name, and click **Edit**. The **Edit Condition Script** dialog box appears.

2. In the **Condition** box, type the expression that you want to use as the condition. You can type the expression to a maximum length of 1024 characters.

3. You can enter a comment in the **Comment** box.

4. In the **Condition Type** list, do one of the following:
   - Click **On False** to configure a script to execute one time when the condition becomes false.
   - Click **While False** to configure a script to execute periodically while the condition is false.
   - Click **On True** to configure a script to execute one time when the condition becomes true.
   - Click **While True** to configure a script to execute periodically while the condition is true.

5. If you selected **While False** or **While True** in the previous step, type a time interval between 1 and 360000 milliseconds in the **Every** box.

   **Note:** The conditional WindowViewer timers will stop themselves if the condition is no longer true. For example, While Mouse Button Down events will not trigger if the mouse button is no longer down, and key scripts will stop if keys are no longer down.

6. Type your script, or modify the existing script in the window.

7. Click **OK**.

To delete all condition scripts that are associated with a condition

- Using the **Classic View**, in the **Scripts** pane, expand **Condition**, right-click the condition script name and click **Delete**. When a message appears, click **Yes**.

To delete individual condition scripts that are associated with a condition

1. Using the **Classic View**, in the **Scripts** pane, expand **Condition**, right-click the key script name and click **Edit**. The **Edit Condition Script** dialog box appears.
2. In the **Condition Type** list, click the script trigger for the script to delete. The script appears in the main section of the **Edit Condition Script** dialog box.

3. On the **Edit** menu, click **Clear**. The script from the main section clears and the associated script is deleted.

### Configuring Data Change Scripts

You can use data change scripts to execute a script one time when a certain tagname or dot field changes by more than its defined dead band.

A data change script is identified by the tagname or tagname field that initiates the script. For example: Tag1 or Tag2.HiHiLimit.

**To configure a data change script**

1. Using the **Classic View**, in the **Scripts** pane, right-click **Data Change** and click **New**. The **Data Change Scripts** dialog box appears.

2. To create a new script, in the **Tagname[.field]** box, enter a tagname or tagname field.
   
   To edit an existing script, click the ellipsis button to the right of the **Tagname[.field]** box and select the script from the list that appears.

3. Type your script in the window.

4. Click **OK**.

**To delete a data change script**

- Using the **Classic View**, in the **Scripts** pane, expand **Data Change**, right-click the data change script name and click **Delete**. When a message appears, click **Yes**.

### Configuring Action Scripts

Use action scripts to associate operator actions with graphic objects. You can configure one or more of the following events with a graphic object:

- Clicking the left, center, or right mouse button.
- Clicking and holding the left, center, or right mouse button.
- Releasing the left, center, or right mouse button.
• Double-clicking the left, center, or right mouse button.
• Pressing a key or key combination.
• Pressing and holding a key or key combination.
• Releasing a key or key combination.
• Moving a mouse pointer over an object.

An action script can only be configured in the Animation Link Selection panel of the object itself.

**Important:** If a key script exists that is triggered by the same key or key combination as the action script, the action script is executed and the key script is ignored.

**To configure an action script**

1. Double-click the graphic object. The Animation Links Selection panel appears.
2. Click **Action**. The **Touch -> Action Script** dialog box appears.

   ![Touch -> Action Script dialog box](image)

3. In the **Condition Type** list, click one of the following:

<table>
<thead>
<tr>
<th>To configure a script that executes on this condition</th>
<th>Click</th>
</tr>
</thead>
<tbody>
<tr>
<td>One time when the left mouse button or a certain key or key combination is pressed</td>
<td><strong>On Left Click/Key Down</strong></td>
</tr>
<tr>
<td>Periodically while the left mouse button or a certain key or key combination is pressed</td>
<td><strong>While Left/Key Down</strong></td>
</tr>
<tr>
<td>One time when the left mouse button or a certain key or key combination is released</td>
<td><strong>On Left/Key Up</strong></td>
</tr>
<tr>
<td>One time when the left mouse button is double-clicked</td>
<td><strong>On Left Double Click</strong></td>
</tr>
<tr>
<td>One time when the right mouse button is pressed</td>
<td><strong>On Right Click</strong></td>
</tr>
<tr>
<td>Periodically while the right mouse button is pressed</td>
<td><strong>While Right Down</strong></td>
</tr>
<tr>
<td>One time when the right mouse button is released</td>
<td><strong>On Right Up</strong></td>
</tr>
<tr>
<td>One time when the right mouse button is double-clicked</td>
<td><strong>On Right Double Click</strong></td>
</tr>
<tr>
<td>One time when the center mouse button is pressed</td>
<td><strong>On Center Click</strong></td>
</tr>
<tr>
<td>Periodically while the center mouse button is pressed</td>
<td><strong>While Center Down</strong></td>
</tr>
<tr>
<td>One time when the center mouse button is released</td>
<td><strong>On Center Up</strong></td>
</tr>
<tr>
<td>One time when the center mouse button is double-clicked</td>
<td><strong>On Center Double Click</strong></td>
</tr>
<tr>
<td>One time when the mouse moves over the object</td>
<td><strong>On Mouse Over</strong></td>
</tr>
</tbody>
</table>

4. If you select **On Left Click/Key Down**, **While Left/Key Down**, or **On Left/Key Up**:
   a. Click **Key**. The **Choose Key** dialog box appears.
b. Click a key.

c. Select the Ctrl and/or Shift check boxes to assign a control key and/or shift key combination to
your selected key.

5. If you select While Left/Key Down or While Right Down, type a time interval between 1 and
360000 milliseconds in the Every box.

6. If you select On Mouse Over, in the After box, type the number of milliseconds between 1 and
360000 to pass after the mouse has moved over the object before the script is executed.

7. Type your script in the window.

8. Click OK.

To delete all action scripts associated with an InTouch graphic object

1. Double-click the graphic object. The object properties panel appears.

2. Click to clear the Action check box. The action scripts will not be executed during run time. If you
click the Action button, the editor opens with the last action script that you saved for any object.

To delete an individual action script

1. Double-click the graphic object that has the action script to delete. The object properties panel
appears.

2. Click the Action button. The Touch -> Action Script dialog box appears.

3. In the Condition Type list, click the script trigger.

4. On the Edit menu, click Clear. The script from the main section clears and the associated script is
deleted.

Configuring ActiveX Event Scripts

Use ActiveX event scripts to run a script when an ActiveX event occurs. Depending on the ActiveX
control, such events can include:

- ActiveX control is started: Startup
- ActiveX control is closed: Shutdown
- User clicks on ActiveX control: Click
- User double-clicks on ActiveX control: Doubleclick
An ActiveX event script is identified by a name. By default, the InTouch HMI automatically adds the control name and event that the script is associated with. For example: MyActiveXScript (AlarmViewerCtrl1::Click).

**To configure a new ActiveX event script**

1. Double-click on the ActiveX control to configure. The ActiveX control properties dialog box appears.

![Control Properties Dialog](image1)

2. Click the **Events** tab.

![Events Tab](image2)

3. Select an event such as click, double-click, shut down, or start up.

4. Click in the **Scripts** cell for that event. Square brackets appear.
5. Type in a new name for an event script and click **OK**. When a message appears, click **OK** to create a new script. The **ActiveX Event Scripts** dialog box appears.

6. In the **Name** box, you can make changes to the ActiveX event script name.

7. Type your script in the window.

8. Click **OK**.

**To edit an existing ActiveX event script**

1. Using the **Classic View**, in the **Scripts** pane, expand **ActiveX Event**, right-click the ActiveX script name and click **Edit**. The **ActiveX Event Scripts** dialog box appears.

2. Make any necessary changes to the script and click **OK**.

**To delete an existing ActiveX event script**

1. Make sure that no ActiveX controls are using the ActiveX event script to delete. If there are ActiveX controls using the script, do the following first:
   a. Remove the ActiveX event script references in the **Events** panel of every ActiveX control that may be using it.
   b. Close all windows and update the use counts.

2. Using the **Classic View**, in the **Scripts** pane, expand **ActiveX Event**, right-click the ActiveX script name and click **Delete**. When a message appears, click **Yes**. The ActiveX event script is deleted.

---

**Pausing Script Execution at Run Time**

By default, when WindowViewer is started, logic is running and synchronous scripts are executed. You can pause script execution at run time by halting logic. After pausing you can resume script execution.

**To pause script execution at run time from the menu**

- On the **Logic** menu, click **Halt Logic**. The synchronous scripts stop running. Asynchronous scripts continue running but no new asynchronous scripts are started.
To pause script execution at run time with scripting

- Write the value 0 to the discrete system tag $LogicRunning. The synchronous scripts stop running. Asynchronous scripts continue running but no new asynchronous scripts are started.

To resume script execution at run time

- On the Logic menu, click Start Logic. The script execution is resumed.

To resume script execution at run time with scripting

- Write the value 1 to the discrete system tag $LogicRunning. The $LogicRunning system tag must be contained in an asynchronous script that is executing at the time the logic is paused.

$LogicRunning System Tag

This system tag monitors and/or controls the running of scripts.

Usage

$LogicRunning

Remarks

Setting the value to 1 starts the script running. Setting the value to 0 stops the script running.

This system tag is equal to selecting Start Logic or Halt Logic on the Logic menu in WindowViewer.

You cannot stop asynchronous scripts that are currently running. However, you can prevent new scripts from running.

Data Type

Discrete (read / write)
About the Script Language

Use these concepts, techniques, and syntax rules for writing scripts using the InTouch HMI script language.

- Basic syntax rules. See Basic Syntax Rules on page 43.
- Calling pre-defined or custom functions. See Calling Standard Functions on page 45 and Calling Custom Functions (QuickFunctions) on page 45.
- Using value assignments and the various operators. See Value Assignments and Operators on page 46.
- Using conditional statements. See Using Conditional Program Branching Structures on page 55.
- Using loops. See Using Program Loops on page 56.
- Using local variables. See Using Local Variables on page 58.

For more information on the general operation of the script editor, see Creating and Editing Scripts on page 13.

For more information on the various types of script triggers, see Script Triggers on page 29.

For a reference of standard script functions, see Built-In Functions on page 65.

Basic Syntax Rules

Basic syntax rules cover these aspects of the InTouch HMI script language:

- Subroutines
- Statements
- Indentation
- Comments
- Tag references
- Literal data values
- Value expressions
Syntax validation

**Subroutines**

There is no concept of separate subroutines within the same script, such as "Sub" procedures in Visual Basic. To structure a script into multiple subroutines, you must create a custom QuickFunction for each subroutine. See *Custom Script Functions* on page 61.

**Statements**

- A statement can be a value assignment, a function call, or a control structure.
- Each statement in a script must end with a semicolon (;).
- You can have multiple statements in the same line, as long as each statement ends with a semicolon.
- You can spread a statement across multiple lines by using line breaks (pressing Enter).

**Indentation**

You can indent your script code in any manner. Indents have no functional relevance.

**Comments**

To mark text as a comment, enclose it in braces `{ }`. Comments can span multiple lines.

**Tag References**

There are several ways to make tag references.

- To refer to a tag that is defined in the local Tagname Dictionary, simply use the tagname.
- To refer to a specific dot field, use the regular reference format *(Tagname.Dotfield)*.
- To refer to a data item on a remote node, use a regular remote tag reference *(AccessName:Item)*.
- You can also define local variables whose scope is limited to the current script. See *Using Local Variables* on page 58.

**Literal Data Values**

- You can specify integer values in decimal or hexadecimal notation. For example, 255 or 0xFF.
- You can specify floating-point values in decimal or scientific notation. For example, 0.001 or 1E-3.
- To specify a Boolean value, use the numerical values 0 for FALSE and 1 for TRUE.
- To specify a string value, enclose it in double quotation marks. For example: "This is a string."

**Value Expressions**

Value expressions can include literal values, tag references and function calls, all linked together by suitable operators. See *Value Assignments and Operators* on page 46.

**Syntax Validation**

When you save a script, the Script Editor automatically checks it for correct syntax. You can also start this validation manually by clicking the **Validate** button. See *Validating Scripts for Correct Syntax* on page 27.
Calling Standard Functions

Standard functions come predefined with the InTouch HMI. See Calling Custom Functions (QuickFunctions) on page 45.

Syntax for Calling Standard Functions

The syntax to call a predefined script function depends on whether and how the function returns a result.

Some functions do not return any result; some functions return an optional result that can be assigned to a tag or used in an expression; some functions return a result that must be assigned to a tag or used in an expression.

To determine the function type, look at the function description. Each function description has a syntax listing that shows whether the function returns a result and whether that result is optional.

To call a function that does not return a result

- Use only the function name (and parameters, if any) in a statement. For example:
  
  FunctionName(Parameters);

To call a function that requires its result to be assigned

- Use the function name (and parameters, if any) anywhere in a script where you could use a literal value or a tagname of the relevant data type. For example, in a value assignment:
  
  ResultsTagname = FunctionName(Parameters);

  Or in a nested function call, using it as a parameter for another standard function:
  
  OtherStandardFunction(FunctionName(Parameters));

To call a function that returns an optional result

- Use either of the preceding procedures.

Passing Parameters to a Function

Parameters to standard predefined functions are usually passed by value. This means that you can pass any valid expression as a parameter, as long as the expression evaluates to the data type that is required for the parameter. Such expressions can include literal values, tag references, and function calls, all linked together by suitable operators. For more information on expressions and operators, see Value Assignments and Operators on page 46.

When the script calls the function, the expression is evaluated and the resulting value passed to the function.

However, there are some functions that require a tag reference as a parameter. For example:

RecipeSelectRecipe(Filename, RecipeName, Number);

In this example, the RecipeName parameter must be a tag reference (that is, you must use a literal tagname for the RecipeName parameter). You cannot pass a string expression instead, even if that expression evaluates to a valid tagname.

Note: Some legacy predefined functions with only one parameter (for example, the Ack() function) do not follow the standard syntax of passing parameters in parentheses. Instead, the parameter is separated from the function name by a space. Check the syntax description in the function documentation if you are in doubt about a particular function.

Calling Custom Functions (QuickFunctions)

Calling a custom QuickFunction differs slightly from calling a predefined standard function:

- The keyword CALL must precede the QuickFunction name.
• Results returned by QuickFunctions are always optional; you can use them, but you do not have to.

To call a QuickFunction that does not return a result

• Use the function name (and parameters, if any) preceded by the keyword CALL in a statement. For example:
  CALL QuickFunctionName(Parameters);

To call a QuickFunction that returns a result

• Do either of the following:
  o Call the QuickFunction as if it did not return a result (see the preceding procedure).
  o Use the function name (and parameters, if any) preceded by the keyword CALL anywhere in a script where you could use a literal value or a tagname of the relevant data type. For example, in a value assignment:
    ResultsTagname = CALL QuickFunctionName(Parameters);

    Or in a nested function call, using it as a parameter for a standard function:
    OtherStandardFunction(CALL FunctionName(Parameters));

Note: You cannot nest QuickFunction calls so that a QuickFunction is used as a parameter for another QuickFunction. For example, CALL QF1(CALL QF2()); is not a valid statement.

Passing Parameters to a QuickFunction

Parameters to QuickFunctions are always passed by value. You cannot pass parameters to QuickFunctions by reference.

You can pass any valid expression as a parameter, as long as the expression evaluates to the data type that is required for the parameter. Such expressions can include literal values, tag references, and function calls, all linked together by suitable operators. For more information on expressions and operators, see Value Assignments and Operators on page 46. When the script calls the function, the expression is evaluated and the resulting value passed to the function.

Note: You cannot nest QuickFunction calls so that a QuickFunction is used as a parameter for another QuickFunction. For example, CALL QF1(CALL QF2()); is not a valid statement.

Value Assignments and Operators

In a script, you use value assignments to write values to a tag. The syntax for a value assignment is as follows:

  Tagname = ValueExpression;

When this statement is executed, ValueExpression is written to the tag referred to by Tagname. ValueExpression can be any valid expression whose data type matches the tag data type. Value expressions can include literal values, tag references, and function calls, all linked together by suitable operators.

See Supported Operators on page 46.

See Setting the Evaluation Order of Operators on page 53.

See Examples for Expressions on page 54.

Supported Operators

The following table lists all supported operators. For information on the use of a specific operator, see the relevant section.
### Operator | More information
--- | ---
+ | Addition or Concatenation: + on page 47
- | Subtraction: - on page 48
* | Multiplication: * on page 48
/ | Division: / on page 48
** | Power: ** on page 48
MOD | Modulo: MOD on page 48
~ | Complement: ~ on page 49
SHL | Shift Left: SHL and Shift Right: SHR on page 49
SHR | Shift Left: SHL and Shift Right: SHR on page 49
& | Bitwise AND: & on page 49
| | Bitwise OR: | on page 50
^ | Bitwise XOR: ^ on page 50
AND | Logical Conjunction: AND on page 51
OR | Logical Disjunction: OR on page 51
NOT | Logical Negation: NOT on page 52
| | Comparisons: <, >, <=, >=, ==, <> on page 52

**Note:** For numeric calculations, always select the operands so that the result of the calculation is still within the value range of a Real number. Otherwise, the result will not be correct.

### Addition or Concatenation: +
Adds two numeric operands or concatenates two string operands.

**Valid operands**
- For addition: Any Integer or Real value
- For concatenation: Any Message value

**Data type of return value**
- For addition: Integer or Real
For concatenation: Message

**Example**

```
MessageTag = "Setpoint value: " + Text(SetpointTag, "#.##");
```

### Subtraction: `-`
When used with two operands, performs a regular numeric subtraction.

**Valid operands**
Any Integer or Real value

**Data type of return value**
Integer or Real

**Example**
In this example, if TemperatureSetpoint is 70, after the script executes TemperatureSetpoint is 65.
```
TemperatureSetpoint = TemperatureSetpoint - 5;
```

### Multiplication: `*`
Regular numeric multiplication.

**Valid operands**
Any Integer or Real value

**Data type of return value**
Integer or Real

### Division: `/`
Regular numeric division. If you try to divide by 0 at run time, 0 is returned as the result.

**Valid operands**
Any Integer or Real value

**Data type of return value**
Integer or Real

### Power: `**`
Raises the left operand (the base) to the power of the right operand (the power).

**Valid operands**
Integer or Real values. It is not possible to combine a base of 0 with a negative power, or a negative base with a fractional power. In these cases, 0 is returned as the result.

**Data type of return value**
Integer or Real

**Example**
```
8 ** (1/3) returns 2 (the cubic root of 8)
```

### Modulo: `MOD`
Returns the remainder of the division of two integer values.
Valid operands
Any Integer value.

Data type of return value
Integer

Example
37 MOD 4 returns 1

Complement: ~
Returns the one’s complement of an integer value. That is, converts each zero-bit to a one-bit and vice versa.

Valid operands
Any Integer value.

Data type of return value
Integer

Shift Left: SHL and Shift Right: SHR
Shifts the binary representation of an integer value to the right or left by a specified number of bit positions. The left operand is the value to be shifted, the right operand is the number of bit positions. Bits shifted out of the word are lost. Bit positions vacated by the shift are set to 0.

Valid operands
Any Integer value.

Data type of return value
Integer

Example
IntTag = IntTag SHL 1; has the following results when executed repeatedly for an initial tag value of 5:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Binary pattern</th>
<th>Tag value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value</td>
<td>0[...]00000101</td>
<td>5</td>
</tr>
<tr>
<td>Execution 1</td>
<td>0[...]00001010</td>
<td>10</td>
</tr>
<tr>
<td>Execution 2</td>
<td>0[...]00010100</td>
<td>20</td>
</tr>
</tbody>
</table>

Bitwise AND: &
Compares the binary representations of two integer numbers, bit for bit, and returns a result according to the following table:
You can use this operator to quickly "mask out" (set to 0) certain parts of a bit pattern. For example, the following statement masks out the upper 24 bits of the `IntTag` tag:

```plaintext
IntTag = IntTag & 255;
```

As shown in the table, the result bit is always 0 if one of the operand bits is 0. In the binary representation of 255, only the lower 8 bits are 1, so the 24 remaining 0-bits cause all the corresponding bits in the result to be set to 0.

**Valid operands**
Any Integer value.

**Data type of return value**
Integer

**Bitwise OR: |**

Compares the binary representations of two integer numbers, bit for bit, and returns a result according to the following table:

<table>
<thead>
<tr>
<th>Bit in first operand</th>
<th>Bit in second operand</th>
<th>Bit in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

This operation is also called "inclusive OR."

**Valid operands**
Any Integer value.

**Data type of return value**
Integer

**Bitwise XOR: ^**

Compares the binary representations of two integer numbers, bit for bit, and returns a result according to the following table:
Logical Disjunction: OR

Returns TRUE if at least one of the discrete operands is TRUE; otherwise, returns FALSE. The truth table for this operator is as follows:

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p OR q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Valid operands
Any Discrete value.

Data type of return value
Discrete

Logical Negation: NOT
Returns TRUE if the discrete operand is FALSE, and vice versa. The truth table for this operator is as follows:

<table>
<thead>
<tr>
<th>p</th>
<th>NOT p</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Valid operands
Any Discrete value.

Data type of return value
Discrete

Comparisons: <, >, <=, >=, ==, <>
These operators compare two values and return TRUE if the condition specified by the operator is met. The operands can be of any data type. For string operands, the comparison is based on alphabetical, non-case-sensitive ordering, with b being greater than a, c greater than b, and so on. For discrete operands, TRUE is considered greater than FALSE. The following table lists all comparison operators along with their conditions:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than</td>
<td>a &lt; b</td>
<td>a is less than b</td>
</tr>
<tr>
<td>Greater than</td>
<td>a &gt; b</td>
<td>a is greater than b</td>
</tr>
<tr>
<td>Less than or equal</td>
<td>a &lt;= b</td>
<td>a is less than or equal to b</td>
</tr>
<tr>
<td>Greater than or equal</td>
<td>a &gt;= b</td>
<td>a is greater than or equal to b</td>
</tr>
<tr>
<td>Equal</td>
<td>a == b</td>
<td>a is equal to b</td>
</tr>
<tr>
<td>Not equal</td>
<td>a &lt;&gt; b</td>
<td>a is not equal to b</td>
</tr>
</tbody>
</table>

Valid operands
Values of any data type (both values must be of the same data type).

Data type of return value
Discrete
Setting the Evaluation Order of Operators

In any expression, you can use parentheses to force operators to be evaluated in a certain order. This works the same way as in any mathematical expression. If you do not use parentheses, your expression is evaluated based on the default precedence rules for operators. The operation with the highest precedence level is executed first, followed by the operation with the second-highest precedence level, and so on.

The following table shows the precedence level of each operator. Operators on the same row have the same precedence level.

<table>
<thead>
<tr>
<th>Operator(s)</th>
<th>Precedence Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>-, NOT, ~</td>
<td>Highest precedence</td>
</tr>
<tr>
<td>**</td>
<td></td>
</tr>
<tr>
<td>*, /, MOD</td>
<td></td>
</tr>
<tr>
<td>+, -</td>
<td></td>
</tr>
<tr>
<td>SHL, SHR</td>
<td></td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, =&gt;</td>
<td></td>
</tr>
<tr>
<td>==, &lt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Lowest precedence</td>
</tr>
</tbody>
</table>

Implicit Data Type Conversion

The InTouch HMI scripting language provides implicit value conversion in assignments between certain data types. However, this can lead to unexpected results, so you should only use this feature with caution.

The following table shows what happens when you assign a value of a certain type to a tag of a different type.

<table>
<thead>
<tr>
<th>Expected data type</th>
<th>Used data type</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discrete</td>
<td>Integer</td>
<td>A value of 0 is interpreted as FALSE. Any other value is interpreted as TRUE.</td>
</tr>
<tr>
<td>Discrete</td>
<td>Real</td>
<td>A value of 0 is interpreted as FALSE. Any other value is interpreted as TRUE.</td>
</tr>
</tbody>
</table>
### Expected data type | Used data type | Remarks
--- | --- | ---
Integer | Discrete | A value of FALSE is converted to 0. A value of TRUE is converted to 1.
Integer | Real | Only the value before the decimal separator is used. All decimal places are discarded.
Real | Discrete | A value of FALSE is converted to 0. A value of TRUE is converted to 1.
Real | Integer | The value is preserved without changes.

For information on using script functions to convert between other data types, see *Converting Data Types* on page 81

### Examples for Expressions

The following table shows some valid expressions, along with the expression’s result and the result’s data type.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Data type of result</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 MOD 4</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>37 MOD 4 == 1</td>
<td>Discrete</td>
<td>TRUE</td>
</tr>
<tr>
<td>NOT (37 MOD 4 == 1)</td>
<td>Discrete</td>
<td>FALSE</td>
</tr>
<tr>
<td>InfoAppActive(InfoAppTitle(&quot;xyz&quot;)) == 1</td>
<td>Discrete</td>
<td>TRUE if a process called &quot;xyz&quot; is running</td>
</tr>
<tr>
<td>&quot;Batch &quot; + Text(IntTag, &quot;000&quot;)</td>
<td>Message</td>
<td>&quot;Batch 010&quot; if IntTag has a value of 10</td>
</tr>
</tbody>
</table>

The following table shows some invalid expressions, along with the reason why they are invalid.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT (37 MOD 4)</td>
<td>NOT requires a discrete operand.</td>
</tr>
<tr>
<td>NOT 37 MOD 4 == 1</td>
<td>NOT has a higher precedence than the other operators, so the InTouch HMI tries to apply NOT to the integer value of 37 instead of the discrete result of the comparison.</td>
</tr>
<tr>
<td>&quot;Batch &quot; + IntTag</td>
<td>When using the + operator to concatenate strings, both operands must be strings.</td>
</tr>
</tbody>
</table>
Using Conditional Program Branching Structures

You can dynamically control the execution path of a script based on certain conditions being met. The InTouch HMI supports IF-THEN-ELSE control structures for this purpose.

The basic syntax for an IF-THEN-ELSE control structure is as follows:

**Syntax**

```
IF Condition THEN
    ... statements and/or another IF-THEN-ELSE structure
[ELSE
    ... statements and/or another IF-THEN-ELSE structure]
ENDIF;
```

Remember the following rules when working with IF-THEN-ELSE structures:

- IF-THEN-ELSE structures can be nested, both in the THEN section and in the ELSE section.
- For every IF statement, there must be a closing ENDIF statement. An ENDIF statement always applies to the nearest prior IF statement on the same nesting level.
- Condition must be a valid discrete expression. The THEN section is executed if **Condition** is TRUE. The ELSE section is executed if **Condition** is FALSE.
- The ELSE section is optional.
- Some other programming languages allow you to check multiple conditions on the same hierarchy level of an IF-THEN-ELSE structure and have one general ELSE section that is executed if all of the conditions evaluate to FALSE. (The If-ElseIf-Else structure in Visual Basic is an example of this.) This is not possible in the InTouch HMI. For every condition to check, you must open a new IF-THEN-ELSE structure. Therefore, to have a single section of code to act as the ELSE code for all conditions, you must place it in the ELSE section of the IF-THEN-ELSE structure at the last nesting level.

**Simple Conditional Structure**

The following script shows a simple conditional structure. If **SuccessTag** is TRUE, the "Success" window opens, otherwise the "Failure" window opens.

```
IF SuccessTag == 1 THEN
    Show "Success";
ELSE
    Show "Failure";
ENDIF;
```

**Nested Conditional Structure**

The following script shows how to check for multiple conditions and have one general ELSE section with code that is executed if none of the conditions are met.

```
IF ChoiceTag == 1 THEN
    Show "Procedure 1";
ELSE
    IF ChoiceTag == 2 THEN
        Show "Procedure 2";
    ELSE
        IF ChoiceTag == 3 THEN
            Show "Procedure 3";
        ELSE
            Show "Default Procedure";
        ENDIF;
    ENDIF;
ENDIF;
```
ENDIF;
ENDIF;

Invalid Scripting Example (Missing ENDIF)

If you are familiar with Visual Basic, you might try to write a simple IF statement like this:

```plaintext
IF OpenThisWindow == 1 THEN Show "This Window";
```

This does not work in the InTouch HMI. For every IF statement, there must be a closing ENDIF statement.

Invalid Scripting Example (Incorrect Nesting)

If you are familiar with a language like Visual Basic, you might want to write a conditional structure with multiple conditions and a default condition like this:

```plaintext
IF ChoiceTag == 1 THEN
  Show "Procedure 1";
ELSE IF ChoiceTag == 2 THEN
  Show "Procedure 2";
ELSE IF ChoiceTag == 3 THEN
  Show "Procedure 3";
ELSE
  Show "Default Procedure";
ENDIF;
```

This does not work in the InTouch HMI. Each IF opens a new nesting level and must have a corresponding ENDIF statement. For a correct version of this example, see Nested Conditional Structure on page 55.

Using Program Loops

Loops allow you to execute a section of code repeatedly. The InTouch HMI only supports FOR loops. A FOR loop works by monitoring the value of a numeric loop variable that is incremented or decremented with each loop iteration. The loop is executed until the value of the loop variable reaches a fixed limit.

**Syntax**

```plaintext
FOR LoopTag = StartExpression TO EndExpression [STEP ChangeExpression] 
  ... statements or another FOR loop ...
NEXT;
```

- StartExpression, EndExpression and ChangeExpression together define the number of iterations.
- StartExpression sets the start value of the loop range. EndExpression sets the end value of the loop range.
- STEP ChangeExpression optionally sets the value by which the loop tag is incremented or decremented during each loop iteration; if you do not specify this, a default of 1 is used.

When you execute a FOR loop, the InTouch HMI:

1. Sets LoopTag to the value of StartExpression.
2. Tests whether LoopTag is greater than EndExpression. If so, the InTouch HMI exits the loop. (If ChangeExpression is negative, the InTouch HMI tests whether LoopTag is less than EndExpression.)
3. Executes the statements within the loop.
4. Increments LoopTag by the value of ChangeExpression (1 unless otherwise specified).
5. Repeats steps 2 through 4.
Remember the following rules when working with **FOR** loops:

- **FOR** loops can be nested. The maximum number of nesting levels depends on the available memory and system resources.
- For every **FOR** statement, there must be a closing **NEXT** statement. A **NEXT** statement always applies to the nearest prior **FOR** statement on the same nesting level.
- **LoopTag** must be a numeric tag (or local variable).
- **StartExpression**, **EndExpression** and **ChangeExpression** must be valid expressions that evaluate to a numeric result.
- If **ChangeExpression** is positive, **EndExpression** must be greater than **StartExpression**; if **ChangeExpression** is negative, **StartExpression** must be greater than **EndExpression**. Otherwise, the loop does not start.
- To exit a loop, use the **EXIT FOR** statement. For more information, see *Forcing the End of a Loop* on page 57.
- There is a time limit for loops. See *Time Limit for Loop Execution* on page 58.

**Caution:** Loop execution affects other run-time processes. For more information, see *Effect of Loops on Other Run-Time Processes* on page 57.

### Forcing the End of a Loop

You can exit a loop at any time by calling the following statement:

```plaintext
EXIT FOR;
```

This statement causes script execution to continue at the statement immediately following the loop **NEXT** statement.

**Example**

The following code fragment uses a loop to insert a large number of dummy records into a database table. If there is an error inserting a record, the loop is aborted to prevent creating more errors.

```plaintext
FOR Counter = 1 TO 1000
    ResultCode = SQLInsert(ConnectionID, "BatchDetails", "BindList1");
    IF ResultCode <> 0 THEN
        LogMessage("Error creating records! Aborting...");
        EXIT FOR;
    ENDIF;
NEXT;
```

### Effect of Loops on Other Run-Time Processes

While a **FOR** loop is executing, all other run-time processes in WindowViewer are paused. This includes the following areas:

- Screen updates (animation links, value displays, trends, etc.). This means that you cannot use **FOR** loops to animate objects, because no movement will occur until after the loop has completed.
- I/O communications. For example, if you modify the value of an I/O tag in a **FOR** loop, only the value after the final iteration is written to the I/O device.
- Other scripts, including asynchronous QuickFunctions.

You can avoid pausing other run-time processes by placing the **FOR** loop in an asynchronous QuickFunction.
Time Limit for Loop Execution

To avoid infinite loops, there is a time limit during which FOR loops must complete execution. If a loop does not complete execution after this time span, WindowViewer automatically terminates it and writes a message about the termination to the Log Viewer.

The default time limit is 5 seconds. You can customize it by adding the following line to the intouch.ini file in your application directory:

```
[Loop]
LoopTimeout=x
```

Replace \( x \) with the time limit in seconds.

**Note:** The time limit is checked only at the `NEXT` statement of the loop. Therefore, the first iteration of the loop is always executed, even if it takes longer than the time limit.

Examples of Loops

The following script uses a simple loop and an indirect tag to reinitialize 100 tags (Tag001 to Tag100) with a value of 0.

```
DIM Counter AS INTEGER;
FOR Counter = 1 TO 100
  IndirectInteger.Name = "Tag" + Text(Counter, "000");
  IndirectInteger.Value = 0;
NEXT;
```

The following script uses two nested loops and an indirect tag to reinitialize 1000 tags (Line01_Tag001 to Line10_Tag100) with a value of 0.

```
DIM LineCounter AS INTEGER;
DIM TagCounter AS INTEGER;
FOR LineCounter = 1 TO 10
  FOR TagCounter = 1 TO 100
    IndirectInteger.Name = "Line" + Text(LineCounter, "00") + "_Tag" +
    Text(TagCounter, "000");
    IndirectInteger.Value = 0;
  NEXT;
NEXT;
```

Using Local Variables

You can declare local variables in a script to store temporary or intermediate results. This increases performance and helps to keep your tag count low. You can use local variables just like tagnames in your script. However, there are certain differences:

- Local variables only exist within the scope of the script in which they are declared. They lose their value when script execution finishes. They cannot be referenced by any other scripts in your application.
- Local variables do not have dotfields.
- Local variables do not count towards the tag count.

Before you can use a local variable in a script, you must declare it; otherwise, the reference is considered a tagname. See **Declaring a Local Variable** on page 59.

You can declare local variables that have the same names as tags. See **Naming Conflicts between Local Variables and Tags** on page 59.
Declaring a Local Variable

You can declare local variables anywhere in your script, as long as you declare them before their first use. To declare a local variable, use the following statement:

```
DIM LocVarName [AS DataType];
```

LocVarName is the name of the local variable. The name must follow the naming conventions for tag names. For more information, see Tag Name Conventions in the InTouch® HMI Data Management Guide.

DataType is the data type of the local variable. Valid values are Discrete, Integer, Real, and Message. If you do not specify this option, Integer is used as the default.

You must use a separate DIM statement for each local variable to declare.

You can declare any number of local variables. The number is only limited by the available memory.

Examples

To declare an Integer variable:

```
DIM MyLocalIntVar AS Integer;
```

To declare multiple Real variables:

```
DIM MyLocalRealVar1 AS Real;
DIM MyLocalRealVar2 AS Real;
```

The following statement is not valid:

```
DIM MyLocalRealVar1, MyLocalRealVar2 AS Real;
```

Naming Conflicts between Local Variables and Tags

You can declare a local variable with the same name as an existing tag. However, when you refer to that name in a script, the local variable always takes precedence over the tag. For example, assume you have an existing Integer tag called "iTag," and you run the following script:

```
DIM iTag as Integer;
iTag = 20;
```

In this scenario, the value assignment writes a value to the local variable only. The value of the tag with the same name remains unchanged.


Custom Script Functions

In This Chapter

About Custom Script Functions ................................................................. 61
About QuickFunctions ............................................................................. 61
Configuring QuickFunctions .................................................................. 61
Calling QuickFunctions .......................................................................... 62
Creating Asynchronous QuickFunctions .................................................. 63

About Custom Script Functions

InTouch HMI QuickFunctions are scripts that in other environments might be known as macros, subroutines, or procedures.

About QuickFunctions

QuickFunctions are scripts that you can call from other scripts and animation links. The main advantage of QuickFunctions is a reduction in duplicate code.

You can pass values to QuickFunctions, which can use the values and return results.

QuickFunctions can run asynchronously. Unlike other scripts, they can run in the background without disrupting the main program flow. A QuickFunction running asynchronously can be used for time-consuming operations, such as SQL database calls.

**Note:** Plan QuickFunctions and their arguments carefully, because if you want to modify the arguments in a QuickFunction, you must first delete all calls to that QuickFunction from every script that uses the QuickFunction. After the change is made, you must then add the QuickFunction call back to the scripts. See the note in *Configuring QuickFunctions* on page 61.

There are three basic parts of a QuickFunction:

- Name
- Arguments (optional)
- Script body with optional return values

QuickFunctions are executed by using the CALL function in either an animation link or another script. See *Calling QuickFunctions* on page 62.

Configuring QuickFunctions

You can create, modify, or delete QuickFunctions.
To create a QuickFunction

1. In the Scripts pane, right-click QuickFunctions, and then click New. The QuickFunctions dialog box appears.

2. In the Function box, enter a name for the QuickFunction.

3. In the Arguments area, for each argument, enter a name on the left and a data type on the right. Arguments are local variables that exist only within the QuickFunction in which they are defined. You can have up to 16 arguments per QuickFunction. Argument names can have 31 characters but no spaces. The argument names must begin with an alpha character. Argument names must be unique.

4. Type your script in the window.

5. To cause the QuickFunction to return a result, add to your script: RETURN value
   Value can be a literal value, a local variable, or global tagname or calculated expression. The script terminates at the RETURN command and continues at the calling function.

6. Click OK.

To modify a QuickFunction

1. In the Scripts pane, expand QuickFunctions, right-click the QuickFunction to modify and click Edit. The QuickFunctions dialog box appears.

2. Make modifications to the script body and click OK.

Note: You cannot make modifications to the argument list if there are calls to the QuickFunction in the InTouch application. You must delete those calls first, close all InTouch windows, and update the use counts.

To delete a QuickFunction

1. Delete all calls to the QuickFunction, close all InTouch windows, and update the use counts.

2. In the Scripts pane, expand QuickFunctions, right-click the QuickFunction to delete and click Delete. When a message appears, click Yes.

Calling QuickFunctions

You can configure scripts and animation links to call QuickFunctions and to process or show a possible return value.
A QuickFunction is not called if the parameter values have not changed. You can use $second as a parameter to insure a QuickFunction is executed at least every second.

For more information, see Calling Custom Functions (QuickFunctions) on page 45.

Creating Asynchronous QuickFunctions

You can define QuickFunctions to run asynchronously (that is, parallel) to the main program flow.

To create an asynchronous QuickFunction

1. In the Script Editor, create a QuickFunction.
2. On the Options menu, click Asynchronous.

Limitations of Asynchronous QuickFunctions

You cannot:

- Return a value from an asynchronous QuickFunction.
- Run more than one instance of the same QuickFunction at the same time.
- Stop asynchronous QuickFunctions after they start executing.

You should not:

- Run more than three different asynchronous QuickFunctions at the same time. Running more than three QuickFunctions at the same time reduces system performance significantly.
- Use asynchronous functions as part of expressions for animation links, e.g. Tool Tips.

Checking if any Asynchronous QuickFunctions are Running

You can check if any asynchronous QuickFunctions are running with the IsAnyAsyncFunctionBusy() function. You can use this function to make the QuickScript that calls an asynchronous QuickFunction wait for all other asynchronous QuickFunctions to complete processing.

IsAnyAsyncFunctionBusy() Function

Returns a discrete value indicating if any asynchronous QuickFunctions are running.

Syntax

\[
\text{result} = \text{IsAnyAsyncFunctionBusy (timeout)}
\]

Arguments

- result
  - The discrete value that indicates if asynchronous QuickFunctions are running with following meaning:
    - 0 = No asynchronous QuickFunctions are running.
    - 1 = Asynchronous QuickFunctions are running.
- timeout
  - The number of seconds to wait before checking if any asynchronous QuickFunctions are running. A literal integer value, integer tagname or integer expression.

Example(s)

Assume you want to connect to several SQL databases using asynchronous QuickFunctions, and you know that it takes 2 minutes to make those connections.

First, execute the asynchronous QuickFunctions to connect to the SQL databases.
Next, use the IsAnyAsyncFunctionBusy(120) function in a QuickScript to allow enough time for SQL to make the connections before completing the QuickFunction.

If after 2 minutes the connections have not been made and the asynchronous QuickFunctions are still busy trying to make the connections, a value of 1 (true) is returned by the IsAnyAsyncFunctionBusy() function.

You can now show an error message telling the operator that the SQL connections were unsuccessful.

The following script implements the scenario:

```cpp
IF IsAnyAsyncFunctionBusy(120) == 1 THEN
  SHOW "SQL Connection Error Dialog";
ENDIF;
```

### Stopping Asynchronous QuickFunctions from Running

You cannot stop asynchronous QuickFunctions after they are started, but you can stop further asynchronous QuickFunctions from being started by stopping the script logic. This affects all QuickScripts in your InTouch application.

For more information on stopping script execution, see *Pausing Script Execution at Run Time* on page 41.
InTouch QuickScript functions allow you to execute commands and logical operations based on specified criteria being met. You can use QuickScript functions by themselves and have them executed whenever a certain condition is met, or use them in animation display links.

Important: This chapter includes legacy InTouch QuickScript functions designed to work only on 32-bit versions of the Windows operating system. These functions should not be included in any InTouch QuickScript designed to run on a 64-bit version of Windows. Notes within this chapter identify these legacy 32-bit only functions.

Forcing Updates in Animation Display Links

If you use QuickScripts in animation links, the animation links are only updated if a tag is associated with them. This tag acts as a trigger whenever its value changes. A good choice is to use the $Second or $Minute system tag to update animation links.

To force an update in an animation display link

1. Open the animation link in the object property window.
2. Add a trigger tag (for example $Second) to the calculation. For example:
   - If the animation link is real or integer, you can multiply the expression with $Second/$Second.
   - If the animation link is string, you can add \text{StringMid}(\\text{$TimeString}, 0, 0)\) to the expression.
   - If the animation link is discrete you can add ($second.00 - $second.00) to the expression.

Mathematical Calculations

The InTouch HMI supports basic mathematical functions that you can use in scripts and in animation links, such as functions to:
- Round and truncate numbers.
- Calculate sine and cosine.
- Calculate logarithms and exponentials.
- Calculate the square root.

**Rounding, Truncating, and Determining Sign**

In a script, you can use the following functions to round numbers, truncate numbers, and determine the sign of numbers:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abs()</td>
<td>Calculate the absolute of a value or expression.</td>
</tr>
<tr>
<td>Int()</td>
<td>Calculate the integer of a value or expression.</td>
</tr>
<tr>
<td>Round()</td>
<td>Round a value or expression.</td>
</tr>
<tr>
<td>Sgn()</td>
<td>Determine the sign (minus, plus, zero) of a value or</td>
</tr>
<tr>
<td></td>
<td>expression.</td>
</tr>
<tr>
<td>Trunc()</td>
<td>Return the decimal point prefix of a value or expression.</td>
</tr>
</tbody>
</table>

**Abs() Function**

Returns the absolute value of a specified number. You can use this to convert a negative number to a positive number.

**Syntax**

```
result = Abs(number)
```

**Parameters**

- `number`
  A literal number, analog tagname, or numeric expression.

**Example(s)**

- `Abs(14)` returns 14.
- `Abs(-7.5)` returns 7.5.

**Int() Function**

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

**Syntax**

```
result = Int(number)
```

**Parameters**

- `number`
  A literal number, analog tagname, or numeric expression.

**Example(s)**

- `Int(4.7)` returns 4.
Int(-4.7) returns -5.

**Note:** For negative real numbers, this function returns an integer that is smaller than the specified number. For example, Int(-4.7) is not -4, but -5. To have the integer part returned, use the Trunc() function. See *Trunc()* Function on page 67.

### Round() Function

Rounds a number to a specified precision. The result is a real number.

**Syntax**

```
result = Round (number, precision)
```

**Parameters**

- `number`  
  A literal number, analog tagname, or numeric expression.

- `precision`  
  The precision to which the number is rounded. Can be a literal number, analog tagname, or numeric expression.

**Example(s)**

- `Round(4.3, 1)` returns 4.
- `Round(4.3, 0.01)` returns 4.30.
- `Round(4.5, 1)` returns 5.
- `Round(-4.5, 1)` returns -4.
- `Round(106, 5)` returns 105.
- `Round(43.7, 0.5)` returns 43.5.

### Sgn() Function

Returns the sign of a number. Use it to determine if a number, tagname, or expression is negative, positive, or zero.

**Syntax**

```
result = Sgn (number)
```

**Parameters**

- `number`  
  A literal number, analog tagname, or numeric expression.

**Example(s)**

- `Sgn(425)` returns 1.
- `Sgn(0)` returns 0.
- `Sgn(-37.3)` returns -1.

### Trunc() Function

Returns the truncated value of a number. The truncated value is the part before a decimal point. Use it to work with the integer part of a real number.

**Syntax**

```
result = Trunc (number)
```
Parameters

    number
    A literal number, analog tagname, or numeric expression.

Example(s)

    Trunc(4.3) returns 4.
    Trunc(-4.3) returns -4.

Note: You can also use this function to work with the fractional part of a number. To return the fractional part of a specified number use the Trunc() function as follows:

    result = number - trunc(number);

Using Trigonometric Functions

In a script, you can use the following functions to do trigonometric calculations.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sin()</td>
<td>Calculate the sine of an angle.</td>
</tr>
<tr>
<td>ArcSin()</td>
<td>Calculate the arcus sine of a value or expression.</td>
</tr>
<tr>
<td>Cos()</td>
<td>Calculate the cosine of an angle.</td>
</tr>
<tr>
<td>ArcCos()</td>
<td>Calculate the arcus cosine of a value or expression.</td>
</tr>
<tr>
<td>Tan()</td>
<td>Calculate the tangent of an angle.</td>
</tr>
<tr>
<td>ArcTan()</td>
<td>Calculate the arcus tangent of a value or expression.</td>
</tr>
</tbody>
</table>

Note: Trigonometric QuickScript functions in the InTouch HMI use angles in degrees (0 - 360). To work with radians instead you must perform the corresponding calculation before passing the parameter to the function or after retrieving the result from the function.

Sin() Function

Returns the sine of a number. For trigonometric functions the number is the angle in degrees.

Syntax

    result = Sin (number)

Parameters

    number
    A literal number, analog tagname, or numeric expression.

Example(s)

    Sin(90) returns 1.
    Sin(0) returns 0.
    Sin(30) returns 0.5.
    100 * Sin (6 * $second) returns a sine wave with an amplitude of 100 and a period of one minute.
**ArcSin() Function**

Returns the arc sine of a number. It is the reciprocal function to the Sin() function. Use the ArcSin() function to calculate the angle from -90 to 90 degrees whose sine is equal to that number.

**Syntax**

```plaintext
result = ArcSin (number)
```

**Parameters**

- **number**
  A literal number, analog tagname, or numeric expression in the range of -1 to 1.

**Example(s)**

- `ArcSin(1)` returns 90.
- `ArcSin(0)` returns 0.
- `ArcSin(0.5)` returns 30.

**Cos() Function**

Returns the cosine of a number. For trigonometric functions the number is the angle in degrees.

**Syntax**

```plaintext
result = Cos (number)
```

**Parameters**

- **number**
  A literal number, analog tagname, or numeric expression.

**Example(s)**

- `Cos(90)` returns 0.
- `Cos(0)` returns 1.
- `Cos(60)` returns 0.5.
- `20 + 50 * Cos(6 * $second)` produces a sine wave oscillating around 20 with an amplitude of 50 and a period of one minute.

**ArcCos() Function**

Returns the arcus cosine of a number. It is the reciprocal function to the Cos() function. Use the ArcCos() function to calculate the angle from 0 to 180 degrees whose cosine is equal to that number.

**Syntax**

```plaintext
result = ArcCos (number)
```

**Parameters**

- **number**
  A literal number, analog tagname, or numeric expression in the range of -1 to 1.

**Example(s)**

- `ArcCos(1)` returns 0.
- `ArcCos(-0.5)` returns 120.
**Tan() Function**

Returns the tangent of a specified number. For trigonometric functions the number is the angle in degrees.

**Syntax**

```
result = Tan (number)
```

**Parameters**

- `number`
  - A literal number, analog tagname, or numeric expression.

**Example(s)**

```
Tan(45) returns 1.
Tan(0) returns 0.
```

**ArcTan() Function**

Returns the arcus tangent of a number. It is the reciprocal function to the Tan() function. Use the ArcTan() function to calculate the angle whose tangent is equal to that number.

**Syntax**

```
result = ArcTan (number)
```

**Parameters**

- `number`
  - A literal number, analog tagname, or numeric expression.

**Example(s)**

```
ArcTan(1) returns 45.
ArcTan(0) returns 0.
```

**Returning the Value of Pi**

In a script, you can use the Pi() function to use the constant Pi in mathematical calculations. The Pi() function is exact to 7 digits after the decimal point.

**Syntax**

```
result = Pi()
```

**Example(s)**

```
Pi() returns 3.1415927.
```

**Calculating Logarithms**

In a script, you can use the following functions to run calculations with logarithms and exponential functions.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log()</td>
<td>Calculate the natural logarithm of a value or expression.</td>
</tr>
<tr>
<td>Exp()</td>
<td>Calculate the exponential of a value or expression.</td>
</tr>
</tbody>
</table>
Use | To
---|---
LogN() | Calculate the logarithm of a value or expression to the base of another value or expression.

Log() Function

Returns the natural logarithm of a specified positive number. This is the reciprocal function to the Exp() function.

**Note:** The natural logarithm of 0 and negative numbers is undefined. If you pass 0 or a negative number to the Log() function, it returns a result of -99.0000000.

**Syntax**

```
result = Log (number)
```

**Parameters**

- `number`
  A positive literal number, analog tagname, or numeric expression.

**Example(s)**

- `Log(100)` returns 4.6051702.
- `Log(1)` returns 0.

Exp() Function

Returns the exponential of a specified number. This is the reciprocal function to the Log() function and is equivalent to e raised to a power.

**Note:** If you pass values outside the range of -88.72 to 88.72 to the Exp() function, it returns a result of -99.0000000.

**Syntax**

```
result = Exp (number)
```

**Parameters**

- `number`
  A literal number, analog tagname, or numeric expression in the range of -88.72 to 88.72.

**Example(s)**

- `Exp(1)` returns 2.7182818.
- `Exp(0)` returns 1.

LogN() Function

Returns the logarithm of a positive number to a specified base. This is the reciprocal function to the base to the power of the logarithm.

**Example(s)**

**Syntax**

```
result = LogN (number, base)
```
Parameters

number
A positive literal number, analog tagname, or numeric expression.

base
A positive literal number, analog tagname, or expression unequal to 1.

Example(s)
LogN(8, 2) returns 3.
LogN(num, btag) returns the logarithm of num to the base btag.

Note: If you pass invalid parameters to the LogN() function, it returns a result of -99.000000.

Calculating the Square Root

In a script, you can use the Sqrt() function to calculate the square root of a specified non-negative number.

Note: If you pass a negative value to the Sqrt() function, it returns a result of -99.000000.

Syntax
result = Sqrt (number)

Parameters

number
A non-negative literal number, analog tagname, or numeric expression

Example(s)
Sqrt(36) returns 6.
Sqrt(perftag) returns the square root of the value held by the tagname perftag.

String Operations

You can use many basic string functions in scripts and animation links. You can use these functions to:

- Return parts of strings.
- Change the case of strings.
- Remove and add spaces to strings.
- Handle ASCII values in strings.
- Search and replace in strings.
- Compare strings with each other.
- Return other information about strings, such as their length.

Returning Parts of Strings

In a script, you can use the StringLeft(), StringMid() and StringRight() functions to return parts of strings.

StringLeft() Function

Returns a specified number of characters from the beginning of a string.

Syntax
result = StringLeft (string, length)
### Parameters

- **string**
  A literal text, message tagname, or string expression.

- **length**
  The numbers of characters to return. A literal number, analog tagname, or numeric expression.

### Example(s)

- `StringLeft("Hello World",5)` returns "Hello".
- `StringLeft("Hello World",20)` returns "Hello World".
- `StringLeft("Hello World",0)` returns "Hello World".

**Note:** If you pass 0 as length to the `StringLeft()` function, it returns the entire string.

### StringRight() Function

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

**Syntax**

```plaintext
result = StringRight (string, length)
```

**Parameters**

- **string**
  A literal text, message tagname, or string expression.

- **length**
  The number of characters to return. A literal number, analog tagname, or numeric expression.

**Example(s)**

- `StringRight("Hello World",5)` returns "World".
- `StringRight("Hello World",20)` returns "Hello World".
- `StringRight("Hello World",0)` returns "Hello World".

**Note:** If you pass 0 as length to the `StringRight()` function, it returns the entire string.

### StringMid() Function

Returns a part of a string. You can specify the starting point and how many characters to return.

**Syntax**

```plaintext
result = StringMid (string, startpos, length)
```

**Parameters**

- **string**
  A literal text, message tagname, or string expression.

- **startpos**
  The starting position in the string. A literal number, analog tagname, or numeric expression.

- **length**
  The number of characters to return. A literal number, analog tagname, or numeric expression.

**Example(s)**

- `StringMid("Hello World",5,4)` returns "o W". 
StringMid("Hello World", 7, 50) returns "World".
StringMid("Hello World", 4, 0) returns "lo World".

Note: If you pass 0 as length to the StringMid() function, it returns the entire string after the starting position.

**Changing Case of Strings**

In a script, you can use the StringLower() and StringUpper() functions to return a specified string in lowercase and uppercase. You can assign the result to the specified string to perform a conversion from upper to lowercase or vice versa.

**StringLower() Function**

Returns the lowercase equivalent of a string.

**Syntax**

```plaintext
result = StringLower (string)
```

**Parameters**

- `string` A literal text, message tagname, or string expression.

**Example(s)**

StringLower("TURBINE") returns "turbine".
StringLower("The Value Is 22.2") returns "the value is 22.2".

mtag = StringLower(mtag) converts the message value of mtag to lowercase.

**StringUpper() Function**

Returns the uppercase equivalent of a string.

**Syntax**

```plaintext
result = StringUpper (string)
```

**Parameters**

- `string` A literal text, message tagname, or string expression.

**Example(s)**

StringUpper("abcd") returns "ABCD".
StringUpper("The Value Is 22.2") returns "THE VALUE IS 22.2".

mtag = StringUpper(mtag) converts the message value of mtag to uppercase.

**Removing Spaces from Strings**

In a script, you can trim leading and trailing spaces (blanks) from strings by using the StringTrim() function. You can use this to remove unwanted spaces from a string, for example after a user input.

**StringTrim() Function**

Rrim leading and trailing spaces (blanks) from strings. You can use this to remove unwanted spaces from a string, for example after a user input.
Syntax

\[
\text{result} = \text{StringTrim} \left( \text{string, trimtype} \right)
\]

Parameters

- **string**
  A literal text, message tagname, or string expression.

- **trimtype**
  A literal value, analog tagname, or numeric expression that determines which spaces to remove:
  - 1 = Leading spaces.
  - 2 = Trailing spaces.
  - 3 = Leading and trailing spaces.

Remarks

This function removes all leading and trailing white spaces from a string. White spaces are spaces (ASCII 0x20) and control characters in the range from ASCII 0x09 to 0x0D.

Example(s)

To remove all spaces in a message tag, `mtag`, with an action script, use the following script:

```plaintext
DIM i AS INTEGER;
DIM tmp AS MESSAGE;
mtag = StringTrim(mtag, 3); {mtag is trimmed}
FOR i = 1 TO StringLen(mtag) {run variable i over the characters of mtag}
  IF StringMid(mtag, i, 1)<>" " THEN {i-th character is not space} tmp = tmp + StringMid(mtag, i, 1); {add that character to tmp}
ENDIF;
NEXT;
mtag = tmp; {pass tmp back to mtag}.
```

Other examples:

- `StringTrim(" Joe ", 1)` returns "Joe".
- `StringTrim(" Joe ", 2)` returns "Joe".

This script removes all spaces from the left and the right of the `mtag` value:

```plaintext
mtag = StringTrim(mtag, 3)
```

Formatting Strings with Spaces

In a script, you can use the `StringSpace()` function to add spaces (blanks) to strings.

Syntax

\[
\text{result} = \text{StringSpace} \left( \text{number} \right)
\]

Parameters

- **number**
  A literal number, numeric tagname, or numeric expression.

Example(s)

- `StringSpace(4)` returns a string consisting of 4 blanks.
- "Pump"+StringSpace(1)+"Station" returns "Pump Station".
Converting Between Characters and ASCII Codes

In a script, you can convert characters of a string to ASCII codes and ASCII codes back to characters by using the StringChar() and StringASCII() functions.

These functions do not support multiple byte character sets. Only characters in the range of 0-255 are supported.

Using ASCII codes is useful if you wish to perform some numeric calculation on a string (for example for encoding a string).

StringChar() Function

Returns a single character corresponding to a specified ASCII code.

**Syntax**

```
result = StringChar (ASCIICode)
```

**Parameters**

- **ASCIICode**
  A literal number, numeric tagname, or numeric expression in the range of 0 to 255.

**Remarks**

This function is very useful for passing control characters to external devices (such as printers or modems) or double quotes to SQL queries.

**Example(s)**

- `StringChar(65)` returns "A".
- This script returns "Hello World" enclosed by double quotes: `StringChar(34)+"Hello World"+StringChar(34)`
- This script returns "Hello World" where both words are separated by a carriage return and a line feed: 
  "Hello"+StringChar(13)+StringChar(10)+"World"

StringASCII() Function

Returns the ASCII code of the first character of a string.

**Syntax**

```
result = StringASCII (string)
```

**Parameters**

- **string**
  A literal string, message tagname, or string expression.

**Example(s)**

- `StringASCII("A")` returns 65.
- `StringASCII("hello world")` returns 104.

Searching and Replacing Text in Strings

For languages that use single-byte character sets (such as English) you can use the StringInString() and StringReplace() functions in a script to perform limited search and replace functionality on message tags.

**Use** To

| StringInString() | Search for a certain string in another string and return the result as a position. |


StringReplace() Function

Replace certain characters or words with other characters or words in a specified string and return the result as a new string.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringReplace()</td>
<td>Replace certain characters or words with other characters or words in a specified string and return the result as a new string.</td>
</tr>
</tbody>
</table>

StringInString() Function

Returns the first position of a specified string in another string.

Syntax

```
result = StringInString (string, searchfor, startPos, caseSens)
```

Parameters

- **string**
  This is the string to searched. A literal string, message tagname, or string expression.

- **searchfor**
  This is the string that is to be searched for. A literal string, message tagname, or string expression.

- **startPos**
  This is the starting position in string of the search. A literal value, numeric tagname, or numeric expression.

- **caseSens**
  Determines whether the search is case sensitive. Can be 0 or 1, discrete tagname, or Boolean expression.
  
  - 0 - search is not case sensitive (uppercase and lowercase are considered the same).
  - 1 - search is case sensitive (uppercase and lowercase are considered to be different).

Remarks

Use this function to determine if a certain string is contained in a message tag. You can specify the starting position for the search and whether the letter case is to be respected.

Example(s)

This script returns 5—because the first "M" in "MTX" is in the fifth position of the string:

```
StringInString("DBO MTX-010","MTX",1,0)
```

This script returns 3—because the first "M" in "MTX" is in the third position in the string:

```
StringInString("T-MTX 010 MTX","MTX",1,0)
```

This script returns 11—because the first "M" in "MTX" after the 8th position is in the 11th position in the string:

```
StringInString("T-MTX 010 MTX","MTX",8,0)
```

This script returns 11—because the first string that matches MTX in the correct case is in the 11th position:

```
StringInString("t-mtx 030 MTX","MTX",1,1)
```

This script returns 0—because there is no "Mty" in the string:

```
StringInString("t-mtx 030 MTY-Mtx","Mty",1,1)
```

StringReplace() Function

Searches for a string within another string and, if found, replaces it with yet another string. You can specify:
- Case-sensitivity - This determines if uppercase letters and lowercase letters are to be treated as identical letters or not.
- Number of occurrences to replace - This is useful if more than one occurrence of the search string is found.
- Match whole words - Use this if the search string is a whole word.

**Note:** This function does not support double byte character sets.

**Syntax**

```plaintext
result = StringReplace (string, searchfor, replacewith, casesens, numtoreplace, matchwholewords)
```

**Parameters**

- `string`
The string to search within. A literal string, message tagname, or string expression.
- `searchfor`
The string that is to be searched for. A literal string, message tagname, or string expression.
- `replacewith`
The string that is used as replacement. A literal string, message tagname, or string expression.
- `casesens`Determines whether the search is case sensitive. Can be 0 or 1, discrete tagname or Boolean expression.
  - 0 - search is not case sensitive (uppercase and lowercase are considered the same)
  - 1 - search is case sensitive (uppercase and lowercase are considered to be different)
- `numtoreplace`
The number of replacements to make. Set it to -1 to replace all occurrences of the found search string. A literal integer value, integer tagname, or integer expression.
- `matchwholewords`
  Determines whether only whole words are matched. Can be 0 or 1, discrete tagname, or Boolean expression.
  - 0 - the function looks for the search string characters anywhere in the string
  - 1 - only whole words are matched

**Example(s)**

This statement replaces only the first occurrence and returns "MTY 030 MTX".

```plaintext
StringReplace("MTX 030 MTX","MTX","MTY",0,1,0)
```

This statement replaces all occurrences and returns "MTY 030 MTY".

```plaintext
StringReplace("MTX 030 MTX","MTX","MTY",0,-1,0)
```

This statement replaces all occurrences that match the case and returns "MTY 030 mtx".

```plaintext
StringReplace("MTX 030 mtx","MTX","MTY",1,-1,0)
```

This statement replaces all occurrences that are whole words and returns "MTY 030 QMTX".

```plaintext
StringReplace("MTX 030 QMTX","MTX","MTY",0,-1,1)
```

**Returning Information about Strings**

In a script, you can use the StringLen() and StringTest() functions to return the length of a specified string and to test whether a character is in a certain group of characters.
**StringLen() Function**

Returns the length of a specified string, including non-visible characters.

**Syntax**

```plaintext
result = StringLen (string)
```

**Parameters**

- `string`
  
  A literal string, message tagname, or string expression.

**Example(s)**

- `StringLength("Twelve percent")` returns 14.
- `StringLength("12%")` returns 3.
- `StringLength("The end." + StringChar(13))` returns 9.

**StringTest() Function**

Tests whether the first character of a string is in a certain group of characters.

**Syntax**

```plaintext
result = StringTest (string, group)
```

**Parameters**

- `string`
  
  A literal string, message tagname, or string expression.

- `group`
  
  The number of the group to test the character against. A literal value, integer tagname, or integer expression in the range of 1 to 11.

1 - alphanumeric characters (A-Z, a-z, 0-9)
2 - numeric characters (0-9)
3 - alphabetic characters (A-Z, a-z)
4 - uppercase characters (A-Z)
5 - lowercase characters (a-z)
6 - punctuation characters (ASCII 0x21 - 0x2F), for example !,@,#,$,%^,&,* and so on
7 - ASCII characters (ASCII 0x00 - 0x7F)
8 - Hexadecimal characters (0-9, A-F, a-f)
9 - Printable characters (ASCII 0x20 - 0x7E)
10 - Control characters (ASCII 0x00 - 0x1F and 0x7F)
11 - White space characters (ASCII 0x09 - 0x0D and 0x20)

**Example(s)**

- This string returns a 1—because "A" is an alphanumeric character:
  
  ```plaintext
  StringTest("ACB123",1)
  ```

- This string returns a 0—because "A" is not a lowercase character:
  
  ```plaintext
  StringTest("ABC123",5)
  ```

**Comparing Strings**

In a script, you can use the `StringCompare()`, `StringCompareNoCase()` and `StringCompareEncrypted()` functions to compare two strings.
<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringCompare()</td>
<td>Make a case-sensitive comparison.</td>
</tr>
<tr>
<td>StringCompareNoCase()</td>
<td>Make a case-insensitive comparison.</td>
</tr>
<tr>
<td>StringCompareEncrypted()</td>
<td>Compare an encrypted string with an unencrypted string.</td>
</tr>
</tbody>
</table>

**StringCompare() Function**

Compares two strings with each other and returns a Boolean result (0 = strings are equal). The case of each letter is respected so that, for example, ‘A’ is considered not equal to ‘a’.

**Syntax**

\[
\text{result} = \text{StringCompare} (\text{string1}, \text{string2})
\]

**Parameters**

- \textit{string1}
  - A literal string, message tagname, or string expression.

- \textit{string2}
  - A literal string, message tagname, or string expression.

**Example(s)**

- StringCompare ("Apple","Apple") returns 0.
- StringCompare ("Apple","apple") returns 1.

This string compares the two message tags and returns a discrete result (0 or 1):

StringCompare (mtag1, mtag2)

**StringCompareNoCase() Function**

Compares two strings with each other and returns an integer result. The case of each letter is not respected so that, for example, ‘A’ is considered equal to ‘a’.

The integer result returns:

- 0 if both strings are identical (ignoring case).
- Non-zero otherwise. The result is the difference of ASCII values between the differentiating character (ignoring case).

**Note:** The result of the StringCompareNoCase() function can be used as a discrete result, as all non-zero values are considered to equal TRUE in InTouch scripting.

**Syntax**

\[
\text{result} = \text{StringCompareNoCase} (\text{string1}, \text{string2})
\]

**Parameters**

- \textit{string1}
  - A literal string, message tagname, or string expression.

- \textit{string2}
  - A literal string, message tagname, or string expression.
Example(s)

This string returns 0—because the strings are considered identical:

```
StringCompareNoCase("Apple","apple")
```

This string returns -6—because the strings are considered not identical and the ASCII values of the first differentiating character "p" minus the ASCII value of the corresponding letter "v" equals -6:

```
StringCompareNoCase("Apple","Avocado")
```

**StringCompareEncrypted() Function**

Compares an encrypted string with an unencrypted string and returns a Boolean result. You can use this function for password verification. For more information on password encryption, see Animating Objects in the InTouch® HMI Visualization Guide.

**Syntax**

```
result = StringCompareEncrypted (plain, encrypted)
```

**Parameters**

- `plain`
  A literal string, message tagname, or string expression.

- `encrypted`
  An encrypted message tagname.

**Example(s)**

This script returns 1 when the plain text and the encrypted text are identical, otherwise it returns 0. Passwd is a message tag containing a value from an encrypted user input. PlainTxt is a message tag against which the user input is to be compared.

```
StringCompareEncrypted(PlainTxt, Passwd)
```

**Converting Data Types**

In a script, you can convert values contained in tag names to other data types by using conversion QuickScripts. This allows you to manipulate string data with mathematical functions or to log values to the ArchestrA® Log Viewer for debugging purposes.

- `Text() Function` on page 81
- `StringFromIntg() Function` on page 82
- `StringFromReal() Function` on page 82
- `StringToIntg() Function` on page 83
- `StringToReal() Function` on page 84
- `DTExt() Function` on page 84

**Text() Function**

The `Text()` function returns the value of a number as a string according to a specified format. You may want to do this to format a value in a certain way or to combine the result with other string values for further processing.

**Syntax**

```
result = Text (number, format)
```

**Parameters**

- `number`
  A literal numeric value, analog tag name, or numeric expression.
format
Use ","0","," or ",."
Use "," to represent a digit, ",. to represent the decimal separator, ",0" to force a leading zero, and ",," to insert a comma.
If you use a zero in the format, it must be followed by zeros. All places to the right of the decimal point must always be zeros. For example, 000.0 is correct, while #0#0.0# is incorrect.
The function rounds the value, if necessary. A literal string, message tagname, or string expression.

Example(s)
Text(66,"#.00") returns "66.00".
Text (1234,"#") returns "1234".
Text (123.4, ",##0.0") returns "123.4".
Text (12.3, "0,000.0") returns "0,012.3".
Text (3.57,"#.#") returns "3.6".
This script returns the string "Reactor Pressure is 1690.3 mbar" if the analog tagname "pressure" contains the value 1690.2743.
"Reactor Pressure is "+Text(pressure,"#.#")+" mbar"

StringFromIntg() Function
In a script, you can convert an integer value to a string value by using the StringFromIntg() function.
This function returns the string value of an integer value and performs a base conversion at the same time. This can be used, for example, to show text together with integer values or for converting integer values to hexadecimal numbers.

Syntax
result = StringFromIntg (number, base)

Parameters
number
A literal integer value, integer tagname, or integer expression.
base
The base of the conversion. This is used for converting the value to a different base, such as binary (2), decimal (10) or hexadecimal (16). A literal integer value, integer tagname, or integer expression.

Example(s)
StringFromIntg(26,2) returns "11010" (binary).
StringFromIntg(26,8) returns "32"—because (base 8: 26 = 3*8 + 2)
StringFromIntg(26,10) returns "26" (decimal).
StringFromIntg(26,16) returns "1A" (hexadecimal).

StringFromReal() Function
In a script, you can convert an real value to a string value by using the StringFromReal() function.
You can also specify to:
• Round the value to a specified precision.
• Pass the value in exponential notation.
This can be used, for example, to show text together with real values or for showing real numbers with exponential notation.

Syntax

```plaintext
result = StringFromReal (number, precision, type)
```

Parameters

- **number**
  A literal value, analog tagname, or numeric expression.

- **precision**
  Specifies how many decimal places are to be used. A literal integer value, integer tagname, or integer expression.

- **type**
  Specifies if the exponential notation is to be used. A literal string, message tagname, or string expression.
  - "f" - Use floating point notation.
  - "e" - Use exponential notation with lowercase "e".
  - "E" - Use exponential notation with uppercase "E".

Example(s)

- `StringFromReal(263.355, 2, "f")` returns "263.36".
- `StringFromReal(263.355, 2, "e")` returns "2.63e2".
- `StringFromReal(263.55, 3, "E")` returns "2.636E2".
- `StringFromReal(0.5723, 2, "E")` returns "5.72E-1".

StringToIntg() Function

In a script, you can convert a value contained in a string to an integer value by using the StringToIntg() function.

You can use this to read a value contained at the beginning of a string into an integer tag for further mathematical operations.

Syntax

```plaintext
result = StringToIntg (string)
```

Parameters

- **string**
  A literal string, message tagname, or string expression.

Remarks

The function checks the first character of the string. If it is a number, it attempts to read this and the following characters as an integer number until a non-numeric character is met. The function ignores leading spaces in the string.

Example(s)

- `StringToIntg("ABCD")` returns 0.
- `StringToIntg("13.4 mbar")` returns 13.
- `StringToIntg("Pressure is 13.4")` returns 0.

To extract the first integer from a string (mtag) that is not at the beginning and to store it in the integer tag itag, use the following action script:
DIM i AS INTEGER;
DIM tmp AS INTEGER;
FOR i = 1 TO StringLen(mtag) {run variable i over the characters of mtag}
  tmp = StringASCII(StringMid(mtag, i, 1)) - 48; {detect ASCII value}
  IF (tmp>=0 AND tmp<10) THEN {if ASCII value represented "0" - "9"}
    itag = StringToIntg(StringMid(mtag, i, 0)); {set itag to value from that position and exit loop}
    EXIT FOR;
  ENDIF;
NEXT;

**StringToReal() Function**

In a script, you can convert a value contained in a string to a real value by using the StringToReal() function.

You can use this to read a value contained at the beginning of a string into a real tag for further mathematical operations.

**Note:** This function also supports the exponential notation and converts a string expression 1e+6 correctly to 1000000.

**Syntax**

result = StringToReal (string)

**Parameters**

*string*

A literal string, message tagname, or string expression.

**Remarks**

The function checks the first character of the string. If it is a number, it attempts to read this and the following characters as a real number until a non-numeric character is met. The function ignores leading spaces in the string.

To extract the first real number from a string (message tag mtag) that is not at the beginning and store it in the real tag rtag1, use the following script:

DIM i AS INTEGER;
DIM tmp AS INTEGER;
FOR i = 1 TO StringLen(mtag) {run variable i over the characters of mtag}
  tmp = StringASCII(StringMid(mtag, i, 1)) - 48; {detect ASCII value}
  IF (tmp>=0 AND tmp<10) THEN {if ASCII value represented "0" - "9"}
    rtag = StringToReal(StringMid(mtag, i, 0)); {set rtag to value from that position and exit loop}
    EXIT FOR;
  ENDIF;
NEXT;

**Example(s)**

StringToReal("ABCD") returns 0.
StringToReal("13.4 mbar") returns 13.4.
StringToReal("Pressure is 13.4") returns 0.

**DTExt() Function**

In a script, you can convert a Boolean value to a string value by using the DTExt() function. You can use this function to use customized message display animation links.

This function returns different string values depending on the value of a Boolean value.
Syntax
\[
\text{result} = \text{Dtext (Boolean, stringtrue, stringfalse)}
\]

Parameters

\textit{Boolean}

A literal Boolean value, discrete tagname, or Boolean expression.

\textit{stringtrue}

The string to be returned if Boolean is true. A literal string value, message tagname, or string expression.

\textit{stringfalse}

The string to be returned if Boolean is false. A literal string value, message tagname, or string expression.

Example(s)

This script returns "Running" if the discrete tagname switch is TRUE, otherwise it returns "Stopped".
\[
\text{DText(switch, "Running", "Stopped")}
\]

This script returns the On and Off Messages of another discrete tag switch2 depending on the value of the discrete tag switch1.
\[
\text{DText(switch1, switch2.OnMsg, switch2.OffMsg)}
\]

Working with InTouch Windows at Run Time

In a script, you can control the behavior and appearance of InTouch windows. You can also write a script using QuickScripts to print individual InTouch windows or the entire screen.

Expose Window Name Property

You can use \texttt{GetWindowName} script function to help the run-time environment reduce scripting necessary to load windows with the current implementation. It enables you to retrieve the name of the window under which the function has been called.

\textbf{GetWindowName()} Function

Retrieves the name of the window under which the function has been called.

Syntax

The syntax of the script function is as follows:
\[
\text{resultcode} = \text{GetWindowName(tagname)};
\]

Resultcode indicates the success or failure of the script function. The resultcode can be a Discrete/Integer/Real data type. Resultcode will be 1 or 0, based on the success or failure of the script function:

- Resultcode is 1 when the script function is called from window context.
- Resultcode is 0 when the script function is called from non-window context.

\textbf{Note}: Configuration of the return value for the script function is optional. This is similar to the existing script functions in InTouch.

Parameter

\textit{TagName}

The tagname is the out parameter for this function. This will be a message tag to retrieve the window name.

The parameter for the function can be any of the following:
• InTouch tag
• Dot field
• Local variable in the script
• Remote tag reference
• Galaxy reference

The default text loaded by the script browser is:

GetWindowName(TagName);

**Note:** "TagName" is a message tag or Remote Tag Reference (RTR) which is of message type.

### Return Value

The `GetWindowName` script function returns the window name with the return value of 1 in the following scenarios:

• Window scripts (All Condition Types)
• Push button action scripts (All Condition Types)

The `GetWindowName` script function returns an empty string with return value of 0 in the following scenarios:

• Application scripts
• Key scripts
• Condition scripts
• Data change scripts
• ActiveX event scripts
• Quick script functions (Synchronous and Asynchronous)

### Showing a List of Open Windows

In a script, you can show a dialog box containing the list of InTouch windows that are currently open using the OpenWindowsList() function.

#### OpenWindowList() Function

Shows a dialog box containing the list of InTouch windows that are currently open.

You can not use this function in an animation link.

**Syntax**

```
[result = ]OpenWindowsList();
```

**Example(s)**

This script opens the **Open Windows List** dialog box and shows all InTouch windows that are currently open.

```
OpenWindowList()
```

**Note:** When the Use In-Memory Window Cache WindowViewer option is enabled, closed windows may appear in the list created by the OpenWindowList() function.
Checking If a Window is Open, Closed, or Exists

In a script, you check if an InTouch window is open, is closed, or does not exist by using the `WindowState()` function.

**WindowState() Function**

Checks if an InTouch window is open, is closed, or does not exist.

**Syntax**

```plaintext
result = WindowState (windowname)
```

**Parameters**

- `windowname` Name of the window. A literal string value, message tagname, or string expression.

**Return Value**

An integer value with the following meaning:

- **0** - InTouch window exists and is currently closed
- **1** - InTouch window exists and is currently open
- **2** - InTouch window does not exist

**Example(s)**

This script returns 0, if the InTouch window `Main` exists, but is not open.

```
WindowState("Main")
```

Opening InTouch Windows

In a script, you can open an InTouch window by using one of the following QuickScript functions:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Show</strong></td>
<td>Open an InTouch window at the position defined in its location settings.</td>
</tr>
<tr>
<td><strong>ShowAt()</strong></td>
<td>Open an InTouch window at a specified position. The opened window is centered on the position. This function can also be used to move an opened window.</td>
</tr>
<tr>
<td><strong>ShowHome</strong></td>
<td>Open the InTouch window(s) you specified in the Home Windows tab in the WindowViewer Properties dialog box and closes any other windows.</td>
</tr>
<tr>
<td><strong>ShowTopLeftAt()</strong></td>
<td>Open an InTouch window at a specified position. The opened window aligns its top left corner to the position. This function can also be used to move an opened window.</td>
</tr>
</tbody>
</table>

**Show() Function**

Opens an InTouch window at its default position.

**Syntax**

```plaintext
Show windowname
```
Parameters

\textit{windowname}

The name of the window to be opened. A literal string value, message tagname, or string expression.

**Example(s)**

This script opens the window Main.

\begin{verbatim}
Show "Main";
\end{verbatim}

This script opens the window with the name that is stored in the \textit{wname} message tag.

\begin{verbatim}
Show wname;
\end{verbatim}

\textbf{ShowAt() Function}

Opens an InTouch window at a specified position. It also can move an already open InTouch window to a specified position. The position is the center point of the window.

**Note:** The window will not be centered if one of its edges is off-screen.

**Syntax**

\begin{verbatim}
ShowAt (windowname, xpos, ypos)
\end{verbatim}

**Parameters**

\textit{windowname}

The name of the window to be opened or moved.

\textit{xpos}

The horizontal position in pixels that the window center is to be moved to. A literal value, analog tagname, or numeric expression.

\textit{ypos}

The vertical position in pixels that the window center is to be moved to. A literal value, analog tagname, or numeric expression.

**Example(s)**

This script opens the window Main so that it is centered at the position x:450, y:130.

\begin{verbatim}
ShowAt("Main",450,130);
\end{verbatim}

This script opens the window called UserDialog and positions it, so that its center is over the center position of the object that called this function (for example a button).

\begin{verbatim}
ShowAt("UserDialog","\$ObjHor,\$ObjVer");
\end{verbatim}

\textbf{ShowHome() Function}

Opens the InTouch window(s) you specified in the Home Windows tab in the WindowViewer Properties dialog box and closes any other windows.

**Syntax**

\begin{verbatim}
ShowHome;
\end{verbatim}

\textbf{ShowTopLeftAt() Function}

Opens an InTouch window at a specified position. Can also be used to move an open window.

**Syntax**

\begin{verbatim}
ShowTopLeftAt (windowname, xpos, ypos)
\end{verbatim}

**Parameters**

The name of the window to be opened or moved.
**WWMoveWindow() Function**

Moves and resizes an opened InTouch window to a specified position and specified size. The new position and new size apply temporarily while the specified window is open.

**Syntax**

\[
\text{WWMoveWindow} \left( \text{windowname}, \text{xpos}, \text{ypos}, \text{xsize}, \text{ysize} \right)
\]

**Parameters**

- **windowname**
  The name of the window to be opened or moved.

- **xpos**
  The horizontal position in pixels that the window left edge is to be moved to. A literal value, analog tag name, or numeric expression.

- **ypos**
  The vertical position in pixels that the window top edge is to be moved to. A literal value, analog tag name, or numeric expression.

- **xsize**
  The horizontal size in pixels for the specified window. A literal value, analog tag name, or numeric expression.

- **ysize**
  The vertical size in pixels for the specified window. A literal value, analog tag name, or numeric expression.

**Example(s)**

- This script opens the window Main so that its top left corner is positioned at x:450, y:130.

```plaintext
ShowTopLeftAt("Main", 450, 130);
```

### Moving and Resizing a Window

In a script, you can move and resize an opened InTouch window with the WWMoveWindow() function. The new position and new size apply temporarily while the specified window is open.

**Hiding InTouch Windows**

In a script, you can hide InTouch windows by using either of the following functions.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hide</td>
<td>Hide a specified window.</td>
</tr>
<tr>
<td>HideSelf</td>
<td>Hide the currently active window.</td>
</tr>
</tbody>
</table>
Hide() Function

Hides (closes) an InTouch window.

**Syntax**

```
Hide windowname;
```

**Parameters**

`windowname`

The name of the window to be hidden. A literal string value, message tagname, or string expression.

**Example(s)**

This script hides the window called UserConfirmation.

```
Hide "UserConfirmation";
```

HideSelf() Function

Hides (closes) the currently active InTouch window.

**Note:** This function can only be used in an action QuickScript.

**Syntax**

```
HideSelf;
```

**Example(s)**

```
HideSelf;
```

Changing the Color of a Window

In a script, you can change the color of an open InTouch window by using the ChangeWindowColor() function.

ChangeWindowColor() Function

Changes the color of an open InTouch window and returns a result code.

**Syntax**

```
Result = ChangeWindowColor (windowname, rValue, gValue, bValue)
```

**Parameters**

`windowname`

The name of the window for which the color is to be changed. A literal string value, message tagname, or string expression.

`rValue`

The intensity of the red color. A literal integer value, integer tagname, or integer expression in the range of 0 to 255.

`gValue`

The intensity of the green color. A literal integer value, integer tagname, or integer expression in the range of 0 to 255.

`bValue`

The intensity of the blue color. A literal integer value, integer tagname, or integer expression in the range of 0 to 255.

**Return Value**

A value with the following meaning:

0 - Failure, window is not defined or RGB value is out of range.
1 - Success.
2 - Failure. The window exists, but it is not open.

**Printing Windows at Run Time**

In a script, you can print individual InTouch windows or the entire WindowViewer screen by using the `PrintWindow()` or `PrintScreen()` functions. You can also set the printer you want to use with the `SetWindowPrinter()` function.

**SetWindowPrinter() Function**

At run time, you can set the printer you want to use with the `SetWindowPrinter()` function.

**Note:** The printer set with this function is also the printer that is used with the `PrintHT()` function.

**Syntax**

```
SetWindowPrinter (printername)
```

**Parameters**

`printername`

The name of the printer, either as network share or as printer name as it appears in its property window. A literal string value, message tagname, or string expression.

**Example(s)**

In this example, PRTSRV1 is the node name and PRT22SW1 is the share name given to the printer.

```
SetWindowPrinter("\\PRTSRV1\PRT22SW1");
```

In this example, Epson LX-300 is the name of the printer as seen in the Properties window of the printer.

```
SetWindowPrinter("Epson LX-300");
```

In this example, MyPrinter is a message tag containing the name of an installed windows printer or the path to a shared network printer.

```
SetWindowPrinter(MyPrinter);
```

**Recommendations for Printing**

The following list contains some issues to consider when printing. These are applicable to printing a single window or printing the WindowViewer screen.

- Open the window(s) to be printed before printing it. Otherwise Windows and ActiveX Controls may not print correctly.
- You cannot print to the same printer that is currently printing alarms.
- Avoid overlapping of windows and objects on the window when printing.
- Use True Type fonts whenever possible. The default InTouch font (System) is not a True Type font.
- For faster printing consider using a white background, fewer objects, and text instead of graphics.
- WindowViewer waits a certain amount of time before the window is sent to the printer queue. During this time, WindowViewer updates any I/O values for that window in the background. To change this waiting time, open the intouch.ini file and change or add the following line (in milliseconds):

```
PrintWindowWait=10000
```

**PrintWindow() Function**

In a script, you can print an InTouch window with the `PrintWindow()` function.
**Note:** Scripts containing the PrintWindow() function cannot print the following ArchestrA graphic controls within an InTouch window: ListBox, DateTimePicker, CalendarControl, EditBox, CheckBox, RadioButtonGroup, ComboBox, AlarmClient or TrendClient.

**Syntax**

```
[result = ] PrintWindow (windowname, leftmargin, topmargin, width, height, options);
```

**Parameters**

- `windowname`
  The name of the window to be printed. A literal string value, message tagname, or string expression.

- `leftmargin`
  Left margin offset (in inches). A literal numeric value, analog tagname, or numeric expression.

- `topmargin`
  Top margin offset (in inches). A literal numeric value, analog tagname, or numeric expression.

- `width`
  Printout width (in inches). Set this value to 0 for largest aspect ratio. A literal numeric value, analog tagname, or numeric expression.

- `height`
  Printout height (in inches). Set this value to 0 for largest aspect ratio. A literal numeric value, analog tagname, or numeric expression.

- `options`
  A discrete value, 0 or 1, that is only used if width and height are 0. A literal Boolean value, discrete tagname or Boolean expression. Set to:

  - 1 - The window is printed with the largest aspect ratio that is an integer multiple of the window size.
  - 0 - The window is printed with the largest aspect ratio that fits on the page.

**Note:** If the window contains a bitmap, set options to 1 to prevent the bitmap from being stretched.

**Return Value**

- 0 - Printing job is not queued successfully, or window does not exist
- 1 - Printing job is queued successfully

**PrintScreen() Function**

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

**Syntax**

```
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)
o 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.

o 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.

o 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.

o 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);
```

**PrintHT() Function**

In a script, you can create a button to print the historical trend by linking it to an action QuickScript that
executes the PrintHT QuickScript function.

Use the PrintWindow() function instead of the PrintHT() function when you want to print the entire window
instead of just the trend chart.

**Note:** Printing the Historical Trend using the Print option or the PrintHT() function will not print the x & y
values. Use PrintWindow() or PrintScreen() to print the x & y values.

**Syntax**

```plaintext
PrintHT(HistTrendTagname);
```

**Parameter**

- **HistTrendTagname**
  The history trend tag name for the history trend to be printed.

**Starting Tag Viewer**

Tag Viewer is a run-time application that allows you to watch and monitor tags and to modify tag values.
For information about Tag Viewer and its use, see the InTouch HMI Tag Viewer Guide.

**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**

```plaintext
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.

**Working with Date and Time Information**

In a script, you can use system tags and QuickScript functions to use system time and date settings in calculations. InTouch scripting also supports calculations involving multiple time zones and Daylight Saving Time.

**Retrieving Numerical Date and Time Information**

In a script, you can use a variety of numerical system tags and one script function to retrieve information on the system time and date. These tags and the script function can be used in other mathematical operations. The following system tags and script functions are available:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Year</td>
<td>Return the current year.</td>
</tr>
<tr>
<td>$Month</td>
<td>Return the current month of the year.</td>
</tr>
<tr>
<td>$Day</td>
<td>Return the current day of the month.</td>
</tr>
<tr>
<td>$Hour</td>
<td>Return the current hour of the day.</td>
</tr>
<tr>
<td>$Minute</td>
<td>Return the current minute of the hour.</td>
</tr>
<tr>
<td>$Second</td>
<td>Return the current second of the minute.</td>
</tr>
<tr>
<td>$Msec</td>
<td>Return the current milliseconds.</td>
</tr>
<tr>
<td>$Time</td>
<td>Return the time in milliseconds that have passed since midnight in the local time zone.</td>
</tr>
<tr>
<td>$Date</td>
<td>Return the number of whole days that have passed since the 1st January 1970 in the local time zone.</td>
</tr>
<tr>
<td>$DateTime</td>
<td>Return the number of days (including fractions of a day) that have passed since the 1st January 1970 in the local time zone.</td>
</tr>
<tr>
<td>DateTimeGMT()</td>
<td>Return the number of days (including fractions of a day) that have passed since the 1st January 1970 in Coordinated Universal Time (UTC).</td>
</tr>
</tbody>
</table>

**$Year System Tag**

Returns the current year number.

**Syntax**

```
$Year
```
**Data Type**
Integer (read only)

**Example(s)**
This script assigns the string "Welcome to xxxx" to the string Welcome where xxxx is the current year.
Welcome = "Welcome to " + StringFromIntg($Year,10)

**$Month System Tag**
Returns the current month number.

**Syntax**
$Month

**Data Type**
Integer (read only)

**Example(s)**
This script assigns the string "October" to the string MonthName if the current month is 10.
IF $Month==10 THEN
    MonthName="October";
ENDIF;

**$Day System Tag**
Returns the current day of the month.

**Syntax**
$Day

**Data Type**
Integer (read only)

**Example(s)**
This script assigns the string "It is a leap year!" to the string Msg2User if the current date is the 29th February.
IF $Day==29 AND $Month==2 THEN
    Msg2User="It is a leap year!";
ENDIF;

**$Hour System Tag**
Returns the current hour of the day.

**Syntax**
$Hour

**Data Type**
Integer (read only)

**Example(s)**
This script checks if it is 8 PM and the backup has not run yet (expressed by the discrete tag BackupAlreadyRun), and if so, calls a QuickFunction script called RunBackup() and sets the BackupAlreadyRun flag to TRUE.
IF $Hour==20 AND BackupAlreadyRun==0 THEN
    CALL RunBackup();
    BackupAlreadyRun=1;
ENDIF;
$Minute System Tag

Returns the current minute of the hour.

**Syntax**

$Minute

**Data Type**

Integer (read only)

**Example(s)**

This script checks if it is 4:50 PM and if so, shows the window with the name Shift End.

```plaintext
IF $Minute==50 AND $Hour==16 THEN
    Show "Shift End";
ENDIF;
```

$Second System Tag

Returns the current second of the minute.

**Syntax**

$Second

**Data Type**

Integer (read only)

**Example(s)**

This script generates a sine wave function with an amplitude of 100 and a period of one minute.

```plaintext
100*Sin(6*$Second)
```

This script generates a series of 0’s and 1’s that change every second.

```plaintext
$second.00
```

$Msec System Tag

Returns the current milliseconds.

**Note:** By default the InTouch updates all tags every 1000 milliseconds. Because of this, the $Msec system tag seems not to change. If you increase the rate of update in the WindowViewer properties, you can see the $Msec tag updating.

**Syntax**

$Msec

**Data Type**

Integer (read only)

$Time System Tag

Returns the number of milliseconds that have passed since midnight in local time.

**Syntax**

$Time

**Data Type**

Integer (read only)

**Example(s)**

This script returns the number of seconds that have passed since midnight.

```plaintext
$Time/1000
```
$Date System Tag

Returns the number of whole days that have passed since the 1st January 1970.

**Syntax**

```
$Date
```

**Data Type**

*Integer (read only)*

**Example(s)**

This script returns the current time.

```
StringFromTime(($Date*86400)+($Time/1000),3);
```

$DateTime System Tag

Returns the number of days (including fractions) that have passed since the 1st January 1970.

**Syntax**

```
$DateTime
```

**Data Type**

*Real (read only)*

**Example(s)**

This script returns the current time.

```
StringFromTime($DateTime*86400,3);
```

DateTimeGMT() Function

Returns the number of days (including fractions of a day) that have passed since the 1st January 1970 in Coordinated Universal Time (UTC).

**Note:** This function cannot be used in animation display links.

**Syntax**

```
result = DateTimeGMT();
```

**Return Value**

Number of days since 1st January 1970 in UTC. A literal real value.

**Example(s)**

This script returns the current date/time in UTC.

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

Retrieving String Date and Time Information

In a script, you can retrieve date and time information as strings. This is useful for showing date or time on the screen or when calculations on whole time/date strings are required.

You can use the following system tags and the script function.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DateString</td>
<td>Return the system date in short format.</td>
</tr>
<tr>
<td>$TimeString</td>
<td>Return the system time.</td>
</tr>
</tbody>
</table>
### Use | To
--- | ---
UTCDateTime | Return the UTC time and/or date and the time zone of the local computer.

#### $DateString System Tag
Returns the system date in short format as defined in the Regional Settings of the local operating system.

**Syntax**

```
$DateString
```

**Data Type**

String (read only)

**Example(s)**

This script may return 4/28/2006 depending on the short date format setting in the Regional Settings of the operating system.

```
$DateString
```

#### $TimeString System Tag
Returns the system time as defined in the Regional Settings of the local operating system.

**Syntax**

```
$TimeString
```

**Data Type**

String (read only)

**Example(s)**

This script may return 02:40:37 PM depending on the time format setting in the Regional Settings of the operating system.

```
$TimeString
```

#### UTCDateTime() Function
Returns the UTC time, the UTC date and time, or the local time zone.

**Syntax**

```
result = UTCDateTime (format)
```

**Parameters**

- **format**
  
  Determines what content is returned. A literal string value, message tagname, or string expression with the following possible values:
  
  UTC_SHORT - the function returns the UTC time
  
  UTC_LONG - the function returns the UTC date and time
  
  UTC_LOCAL - the function returns the name of the time zone as set in the time zone settings of the local operating system
  
  Any other values return the UTC date and time in default format (ddd mm dd hh:mm:ss yyy).
Example(s)
At 09:24 AM Monday January 6th 2003 in the Pacific time zone, the UTCDateTime() function returns the following.

This script returns 17:24:05
UTCDateTime("UTC_SHORT")

This script returns 01/06/2003 17:24:05
UTCDateTime("UTC_LONG")

This script returns Pacific Standard Time -8:0: 1
UTCDateTime("UTC_LOCAL")

This script returns Mon Jan 06 17:24:05 2003.
UTCDateTime("Invalid")

Converting Date and Time Information to Strings
In a script, you can convert date and time information to strings for easier interpretation and display requirements. You can use the following functions.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringFromTime()</td>
<td>Convert a UTC timestamp to local time and to return as a time string.</td>
</tr>
<tr>
<td>wwStringFromTime()</td>
<td>Convert a local time timestamp to UTC time and returns it as a time string.</td>
</tr>
<tr>
<td>StringFromTimeLocal()</td>
<td>Convert a timestamp as a time string.</td>
</tr>
</tbody>
</table>

StringFromTime() Function
Converts a timestamp given in UTC time to local time and returns the result as a string. This function takes Daylight Saving Time into account.

Note: This function is equivalent to the StringFromGMTTimeToLocal() function.

Syntax
result = StringFromTime (timestamp, format)

Parameters

(timestamp)
The number of seconds that have passed since midnight of the 1st January 1970 in the UTC time zone. A literal integer value, integer tagname, or integer expression.

(format)
Determines how the string result is shown. A literal integer value, integer tagname, or integer expression in the range from 1 to 5 with following meaning:

1 - Shows the date according to the format set in the Regional Settings of the local operating system
2 - Shows the time according to the format set in the Regional Settings of the local operating system
3 - Shows the date and time as a 24 character string (ddd mmm dd hh:mm:ss yyy)
4 - Shows the day of the week in short form
5 - Shows the day of the week in long form
Example(s)

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

<table>
<thead>
<tr>
<th>Script</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StringFromTime(86400,1)</code></td>
<td>1/1/70</td>
</tr>
<tr>
<td><code>StringFromTime(86400,2)</code></td>
<td>04:00:00 PM</td>
</tr>
<tr>
<td><code>StringFromTime(86400,3)</code></td>
<td>Thu Jan 01 16:00:00 1970</td>
</tr>
<tr>
<td><code>StringFromTime(86400,4)</code></td>
<td>Thu</td>
</tr>
<tr>
<td><code>StringFromTime(86400,5)</code></td>
<td>Thursday</td>
</tr>
</tbody>
</table>

wwStringFromTime() Function

Converts a timestamp given in local time to UTC time and returns the result as a string. This function takes Daylight Saving Time into account.

Syntax

```
result = wwStringFromTime (timestamp, format)
```

Parameters

- `timestamp`
  - The number of seconds that have passed since midnight of January 1, 1970 in the local time zone. A literal integer value, integer tagname, or integer expression.

- `format`
  - Determines how the string result is shown. A literal integer value, integer tagname, or integer expression in the range from 1 to 5 with following meaning:
    - 1 - Shows the date according to the format set in the Regional Settings of the local operating system
    - 2 - Shows the time according to the format set in the Regional Settings of the local operating system
    - 3 - Shows the date and time as a 24 character string (ddd mmm dd hh:mm:ss yyyy)
    - 4 - Shows the day of the week in short form
    - 5 - Shows the day of the week in long form

Example(s)

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

<table>
<thead>
<tr>
<th>Script</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wwStringFromTime(57600,1)</code></td>
<td>1/2/70</td>
</tr>
<tr>
<td><code>wwStringFromTime(57600,2)</code></td>
<td>12:00:00 AM</td>
</tr>
<tr>
<td><code>wwStringFromTime(57600,3)</code></td>
<td>Fri Jan 02 00:00:00 1970</td>
</tr>
<tr>
<td><code>wwStringFromTime(57600,4)</code></td>
<td>Fri</td>
</tr>
</tbody>
</table>
This script returns "Friday"
wwStringFromTime(57600,5)

**StringFromTimeLocal() Function**

Converts a timestamp to a time and returns the result as a string.

**Syntax**

```plaintext
result = StringFromTimeLocal (timestamp, format)
```

**Parameters**

- `timestamp`
  The number of seconds that have passed since midnight of January 1, 1970. A literal integer value, integer tagname, or integer expression.

- `format`
  Determines how the string result is shown. A literal integer value, integer tagname, or integer expression in the range from 1 to 5 with following meaning:
  - 1 - Shows the date according to the format set in the Regional Settings of the local operating system
  - 2 - Shows the time according to the format set in the Regional Settings of the local operating system
  - 3 - Shows the date and time as a 24 character string (ddd mmm dd hh:mm:ss yyyy)
  - 4 - Shows the day of the week in short form
  - 5 - Shows the day of the week in long form

**Example(s)**

- This script returns "1/2/70"
  StringFromTimeLocal(86400,1)

- This script returns "12:00:00 AM"
  StringFromTimeLocal(86400,2)

- This script returns "Fri Jan 02 00:00:00 1970"
  StringFromTimeLocal(86400,3)

- This script returns "Fri"
  StringFromTimeLocal(86400,4)

- This script returns "Friday"
  StringFromTimeLocal(86400,5)

**Checking the Daylight Savings Time Status**

In a script, you can check if daylight savings time is active by using the `wwIsDaylightSaving()` function.

**wwIsDaylightSaving() Function**

Returns whether daylight savings time is currently active.

**Syntax**

```plaintext
result = wwIsDaylightSaving()
```

**Return Value**

A Boolean value with following meaning:

- 0 - Daylight savings time is not active.
- 1 - Daylight savings time is active.
Interacting with Other Applications

In a script, you can interact with other Windows applications by using various QuickScripts. For example, you can:

- Start an application, such as Notepad.
- Check an application title name.
- Check if a certain application is running.
- Activate a running application.
- Simulate keyboard strokes.
- Close, minimize or maximize an application window.
- Execute commands and exchange data with applications that support DDE.

Starting a Windows Application

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

**Syntax**

```
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"
```

Retrieving the Application Title of a Running Application

In a script, you can find the application title or Windows task list name of a specified running application by using the InfoAppTitle() function. This information is, for example, required by InTouch scripting for checking if the specified application is currently running or for activating it.

**Note:** This function will not return a value for Internet Explorer. As a workaround, use the Google Chrome browser.

**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")

Checking If an Application is Running

In a script, you can check if a specific application is already running by using the InfoAppActive() function. You need to know the application title or Windows task list name first to be able to check if the specific application is running.

InfoAppActive() Function

Returns the running status of an application.

Syntax
result = InfoAppActive (apptitle)

Parameters

apptitle
The application title or Windows task list of the application for which you want to query the running status. A literal string value, message tagname, or string expression.

Return Value

A Boolean value indicating:

0 - The application is not running
1 - The application is running

Example(s)

This script queries for the application Notepad, and if it is already running, activates it. Otherwise it launches a new instance of Notepad. This way launching Notepad multiple times is avoided.
IF InfoAppActive(InfoAppTitle("Notepad"))==1
THEN
  ActivateApp InfoAppTitle( "Notepad" );
ELSE
  StartApp "Notepad";
ENDIF;

Activating a Running Windows Application

In a script, you can activate a running Windows application by using the ActivateApp() function. This brings the specified application to the foreground and gives it focus.

You need to do the following before activating a running Windows application:

- Find the application title or Windows task list name. See Retrieving the Application Title of a Running Application on page 102.
- Ensure the Windows application is running. See Checking If an Application is Running on page 103.

ActivateApp Function

Activates an already running Windows application.
**Important:** The ActivateApp() function does not work on 64-bit versions of the Windows operating system.

**Syntax**

```plaintext
ActivateApp apptitle;
```

**Parameters**

`apptitle`

The application title or Windows task list name of the running application you want to activate.

**Example(s)**

This script checks if a command prompt window is already open, and if so, activates it. Otherwise it starts
the command prompt window.

```plaintext
IF InfoAppActive( InfoAppTitle("cmd")) == 1 THEN
  ActivateApp InfoAppTitle("cmd");
ELSE
  StartApp "cmd";
ENDIF;
```

---

**Sending Simulated Key Strokes to an Application**

In a script, you can simulate pressing a sequence of keys on the keyboard. You can use this, for
example, to:

- Enter data automatically in an open application.
- Control any application (including the InTouch HMI).

**SendKeys Function**

Simulates a sequence of key strokes.

**Important:** The SendKeys() function does not work on 64-bit versions of the Windows operating system.

**Syntax**

```plaintext
SendKeys sequence;
```

**Parameters**

`sequence`

The sequence of keys strokes to be simulated. A literal string value, message tagname, or string
expression.

In addition to regular characters on the keyboard (such as alphanumeric characters) you can also
specify control keys as a code:

- `{BACKSPACE}` - Simulates the Backspace key
- `{BREAK}` - Simulates the Break key
- `{CAPSLOCK}` - Simulates the Caps Lock key
- `{DELETE}` - Simulates the Delete key (or `{DEL}`)
- `{DOWN}` - Simulates Arrow Down key
- `{END}` - Simulates the End key
- `{ENTER}` - Simulates the Enter key (or `~`)
- `{ESCAPE}` - Simulates the ESC key (or `{ESC}`)
- `{F1} .. `{F12}` - Simulate the F1 .. F12 keys
- `{HOME}` - Simulates the Home key
- `{INSERT}` - Simulates the Insert key
- `{LEFT}` - Simulates the Arrow Left key
- `{NUMLOCK}` - Simulates the Num Lock key
- `{PGDN}` - Simulates the Page Down Key
- `{PGUP}` - Simulates the Page Up key
{PRTSC} - Simulates the Print Screen key
{RIGHT} - Simulates the Arrow Right key
{TAB} - Simulates the Tab key
{UP} - Simulates the Up key
+ - Simulates the Shift key
use with parenthesis surrounding the key(s) you want to press in combination with the Shift key.
^ - Simulates the Ctrl key
use with parenthesis surrounding the key(s) you want to press in combination with the Ctrl key.
% - Simulates the Alt key
use with parenthesis surrounding the key(s) you want to press in combination with the Alt key.

Remarks

Use the StartApp and/or ActivateApp() commands to activate another application before sending simulated keys strokes to it.

Example(s)

This script simulates pressing the B key.
SendKeys "b";

This script simulates pressing the key combination Ctrl and P, which can be used to initiate the Printing dialog box in another application.
SendKeys "^(p)";

This script simulates pressing F1 (which may open the help function), pressing the Tab key (which may place the cursor in a search field), entering HAL, and pressing the Enter key (which may initiate the search).
SendKeys "{F1}{TAB}HAL{ENTER}";

This script simulates pressing Ctrl, Shift and the key 1, which is the same as switching to WindowMaker. This powerful combination can be used for developing self-modifying (dynamic) InTouch HMI applications.
SendKeys "^(+(1))";

Closing, Minimizing or Maximizing a Windows Application

In a script, you can close, minimize, or maximize another Windows application by using the WWControl() command.

You need to do the following before closing, minimizing or maximizing a Windows application:

- Find its application title or Windows task list name. See Retrieving the Application Title of a Running Application on page 102.
- Make sure that the Windows application is running. See Checking If an Application is Running on page 103.

WWControl() Function

Restores, minimizes, maximizes, or closes a Windows application.

Syntax

WWControl (apptitle, control);

Parameters

apptitle
The application title or Windows task list name of the running application you want to restore, minimize, maximize or close. A literal string value, message tagname, or string expression.
**control**
Determined the action you want to take on the specified Windows application. A literal string value, message tagname, or string expression with following values:
- Restore - activates and shows the application window
- Minimize - activates and minimizes the application window
- Maximize - activates and maximizes the application window
- Close - closes the application

**Remarks**
To use this function in Windows Server 2003, you must be a member of the Administrators group, the Performance Log Users group, or the Performance Monitor Users group on the local computer or you must have been delegated the appropriate authority to write to the registry.

**Example(s)**
This script restores the calculator application if it is already running.

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

This script closes the WindowViewer.

```plaintext
WWControl (InfoAppTitle(“View”),"Close");
```

## Executing Commands and Exchanging Data using DDE

You can write a script to interact with applications that support DDE.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWExecute()</td>
<td>Send and execute commands.</td>
</tr>
<tr>
<td>WWRequest()</td>
<td>Read data from DDE items.</td>
</tr>
<tr>
<td>WWPoke()</td>
<td>Write data to DDE items.</td>
</tr>
</tbody>
</table>

## WWExecute() Function

Sends a command to an application, executes it, and returns a status result. You can use it to have Excel to run a macro.

**Important:** The WWExecute() function does not work on 64-bit versions of the Windows operating system.

**Syntax**

```plaintext
Result = WWExecute (appname, topic, command)
```

**Parameters**

- **appname**
  The name of the application the command is sent to. A literal string value, message tagname, or string expression.

- **topic**
  The name of the topic within the application that the command is sent to. A literal string value, message tagname, or string expression.

- **command**
  The command to be sent. A literal string value, message tagname, or string expression.
Return Value

A value of -1, 0, or 1 indicating the following:
-1 - command not executed successfully. Possible causes are the application not running, the topic
does not exist or the command contains an error.
0 - command not executed successfully because the application is busy.
1 - command executed successfully.

Example(s)

This script instructs Microsoft Excel to execute the macro Macro1 by sending the command
[Run("Macro1",0)] to Excel.

```plaintext
Macro="Macro1";
Command="[Run(" + StringChar(34) + Macro + StringChar(34) + ",0)]";
WWExecute("excel","system",Command);
```

WWRequest() Function

Reads data from an item of an application. You can use it, for example, to read the value of a
spreadsheet cell in Microsoft Excel.

**Important:** The WWRequest() function does not work on 64-bit versions of the Windows operating
system.

**Syntax**

```
Result = WWRequest(appname, topic, item, messagetag)
```

**Parameters**

- **appname**
  The name of the application. A literal string value, message tagname, or string expression.

- **topic**
  The name of the topic within the application. A literal string value, message tagname, or string
  expression.

- **item**
  The name of the item belonging to the topic and application. A literal string value, message tagname,
or string expression.

- **messagetag**
  A message tagname to retrieve the value of the item. The message tagname value can be converted
  into an integer or real value by using the StringToIntg() or StringToReal() functions.

**Return Value**

A value of -1, 0, or 1 indicating the following:
-1 - data not read successfully. Possible causes are the application not running or the topic or item do
not exist.
0 - data not read successfully because the application is busy.
1 - data read successfully.

**Example(s)**

This script reads the value contained in Microsoft Excel book Book1.xls, sheet Sheet1 in Row 1, Column
1 to the message tagname MTag and puts the value in the real tagname CellValue.

```plaintext
Result = WWRequest("excel","[Book1.xls]sheet1", "r1c1",Mtag);
CellValue=StringToReal(MTag);
```

If you are using a non-English operating system, you may need to use the StringReplace() function to
change the contents of MTag before converting it to a different data type. For example, for operating
systems that use a comma as a decimal separator, you may need to replace all commas with decimal
dots in MTag before converting it to a real data type.
**WWPoke() Function**

Writes data to an item of an application. You can use it, for example, to write the value into a spreadsheet cell in Excel.

---

**Important:** The WWPoke() function does not work on 64-bit versions of the Windows operating system.

---

**Syntax**

```plaintext
result = WWPoke (appname, topic, item, string)
```

**Parameters**

- **appname**
  The name of the application. A literal string value, message tagname, or string expression.

- **topic**
  The name of the topic within the application. A literal string value, message tagname, or string expression.

- **item**
  The item name belonging to the topic and application. A literal string value, message tagname, or string expression.

- **string**
  The value to be written. A literal string value, message tagname, or string expression. You can use the StringFromIntg(), StringFromReal() or Text() functions to convert the value of an integer or real tagname to a message tagname.

---

**Return Value**

A value of -1, 0, or 1 indicating the following:

- -1 - data not written successfully. Possible causes are the application not running or the topic or item do not exist.
- 0 - data not written successfully because the application is busy.
- 1 - data written successfully.

**Remarks**

Do not use the WWPoke() or WWRequest() function to read and write data between InTouch applications on different nodes or sessions. To read and write data between InTouch applications, use Access Names instead. See Setting Up Access Names in the InTouch® HMI Data Management Guide.

**Example(s)**

This script puts the value of the real tagname CellValue in the message tagname Mtag and writes the value to the spreadsheet cell Row 1, Column 1 of sheet Sheet1 in Microsoft Excel book Book1.xls.

```plaintext
MTag = Text(CellValue, "0");
Result = WWPoke("excel", "[Book1.xls]sheet1", "r1c1", Mtag);
```

---

**Working with Files**

You can write a script using various file management and access operations.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileCopy()</td>
<td>Copy files.</td>
</tr>
<tr>
<td>FileDelete()</td>
<td>Delete files.</td>
</tr>
<tr>
<td>FileMove()</td>
<td>Move files.</td>
</tr>
</tbody>
</table>
Use                      To
----------------------------------
FileReadFields(), FileWriteFields()  Read/write csv data.
FileReadMessage(), FileWriteMessage()  Read/write text data.

Managing Files

In a script, you can copy, delete or move files.

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

result = FileCopy (sourcefile, destfile, progresstag)

Parameters

sourcefile
Full path and file name of the file to be copied. A literal string value, message tag name, 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 tag name, 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 copy progress. A literal string value, message tag name (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.

\[\text{Status} = \text{FileCopy("c:\MyData\output.log","d:\archive\output.txt","Monitor")};\]

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

\[\text{Status} = \text{FileCopy("c:\*\.txt", "c:\Backup", "Monitor")};\]

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.

\[\text{Status} = \text{FileCopy(LogFile, "c:\results\log" + $DateString + $TimeString + ".txt", "Monitor")};\]

FileDelete() Function

Deletes an individual file.

Syntax

\[\text{result} = \text{FileDelete (filename)}\]

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.

\[\text{Status} = \text{FileDelete("c:\Data.txt")};\]

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

\[
\text{result} = \text{FileMove} (\text{sourcefile}, \text{destfile}, \text{progresstag})
\]

Parameters

- **sourcefile**
  - Full path and file name of the file to be moved. A literal string value, message tag name, 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 tag name, 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 tag name (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.

\[
\text{Status}=\text{FileMove} ("c:\MyData\output.log", "d:\archive\output.txt", "Monitor");
\]

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

\[
\text{Status}=\text{FileMove} ("c:\*\*.txt", "c:\Backup", "Monitor");
\]

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.

\[
\text{Status}=\text{FileMove} (\text{LogFile}, "c:\results\log" + \$\text{DateString} + \$\text{TimeString} + ".txt", "Monitor");
\]

Reading and Writing CSV Data

You can write a script to read and write data contained in a csv (comma separated variable) file from and to a series of tagnames by using the function FileReadFields() and FileWriteFields().

The functions FileReadFields() and FileWriteFields() support only the comma as a delimiter.

**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**

\[\text{result } = \text{ FileReadFields (filename, offset, starttag, numberoffields)}\]

**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.

- **numberoffields**
  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.

\[\text{FileReadFields ("c:\set.csv",0,"RecipeTag1",4)}; \]

**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**

\[\text{result } = \text{ FileWriteFields (filename, offset, starttag, numberoffields)}\]

**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.

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

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

```
Flour,27.23,14,1
```

### Reading and Writing Text Data
You can write a script to read and write text data to and from a file by using the FileReadMessage() and FileWriteMessage() functions. You can either read/write a specified number of bytes or an entire line of text (demarcated by a line feed character).

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

**Syntax**

```plaintext
[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`.

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

**FileWriteMessage() Function**

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

**Syntax**

```plaintext
[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`
  Specifies 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`.

```plaintext
FileWriteMessage("c:\Data\File.txt", -1, MsgTag, 1);
```

**Retrieving System-Related Information**

In a script, you can retrieve system-related information using the following QuickFunctions.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetNodeName()</td>
<td>Retrieve the node name of the computer.</td>
</tr>
<tr>
<td>InfoDisk()</td>
<td>Retrieve disk space information.</td>
</tr>
<tr>
<td>InfoFile()</td>
<td>Retrieve information about a file.</td>
</tr>
<tr>
<td>InfoResources()</td>
<td>Retrieve information about the Windows environment.</td>
</tr>
</tbody>
</table>
Retrieving the Node Name of the Computer

In a script, you can retrieve the node name of the computer with the GetNodeName() function. This can be used, for example, to keep your InTouch applications dynamic when working with access names.

GetNodeName() Function

Returns the node name of the computer.

Syntax

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.

```plaintext
GetNodeName(NodeName,131);
GetNodeName("InTouch:NodeName",131);
```

Retrieving Disk Space Information

In a script, you can retrieve disk space information by using the InfoDisk() function. You can retrieve:

- The total size of the disk drive (in bytes or kilobytes).
- The available free space on the disk drive (in bytes or kilobytes).

You can also determine when or how often the information updates (in an animation link) by specifying a trigger tag.

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 tagname, 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.

```
InfoDisk("C", 4, $Minute)
```

### Retrieving Information on a File or Directory
In a script, you can retrieve information on a specific file or directory by using the `InfoFile()` function. By using different parameters you can find:

- If the file exists.
- If the specified file name is actually a directory.
- The size (in bytes) of the file.
- The timestamp of the file or directory.
- The number of files that match a wildcard search.

### InfoFile() Function
Returns various information on a file or directory.

**Syntax**

```
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 tagname, 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.

```plaintext
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.

```plaintext
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.

```plaintext
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.

```plaintext
InfoFile("c:\data\*.txt",4,$minute)
```

**Retrieving Information on the Windows Environment**

In a script, you can retrieve information on the Windows environment by using the InfoResources() function. You can find:

- The free bytes of the paging file.
- The approximate number of Windows tasks.

**InfoResources() Function**

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

**Syntax**

```plaintext
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.

```plaintext
InfoResources(2,$second);
```
Retrieving InTouch Related Information

In a script, you can retrieve InTouch related information using these functions.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfoInTouchAppDir()</td>
<td>Get information on the directory of the InTouch application you are developing.</td>
</tr>
<tr>
<td>InTouchVersion()</td>
<td>Get information on the InTouch version.</td>
</tr>
</tbody>
</table>

Retrieving the Name of the InTouch Application Directory

In a script, you can retrieve the name of the directory that your InTouch application is running in with the InfoInTouchAppDir() function. This function is useful to locate any external files that you include to ship with your InTouch application.

InfoInTouchAppDir() Function

Returns the current InTouch application directory.

Syntax

\[
\text{result} = \text{InfoInTouchAppDir}();
\]

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()

Retrieving the InTouch Version

In a script, you can retrieve the version number of the InTouch application you are currently running by using the InTouchVersion() function.

InTouchVersion() Function

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

Syntax

\[
\text{result} = \text{InTouchVersion} (\text{infotype});
\]

Parameters

- \text{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>

**Security-Related Scripting**

You can add and manage security within your InTouch application with various QuickScript functions and system tags. For more information about security functions, see Securing InTouch in the InTouch® HMI Application Management and Extension Guide.

**Logging On and Off**

You can use the following functions and system tags to log on and log off.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttemptInvisibleLogon()</td>
<td>Log on a user by supplying authentication data in the parameters.</td>
</tr>
<tr>
<td>LogonCurrentUser()</td>
<td>Log on the currently logged on Windows user (if authentication mode is &quot;OS&quot;).</td>
</tr>
<tr>
<td>PostLogonDialog()</td>
<td>Show the <strong>Logon</strong> dialog box.</td>
</tr>
<tr>
<td>Logoff()</td>
<td>Log off the current user.</td>
</tr>
<tr>
<td>$PasswordEntered</td>
<td>Set a password.</td>
</tr>
<tr>
<td>$OperatorEntered</td>
<td>Set a valid user name.</td>
</tr>
<tr>
<td>$OperatorDomainEntered</td>
<td>Set a valid user domain name (if authentication mode is &quot;OS&quot;).</td>
</tr>
</tbody>
</table>

For more information about security functions, see Securing InTouch in the InTouch® HMI Application Management and Extension Guide.
## Changing and Setting Password

You can use the following functions and system tags to change password:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangePassword()</td>
<td>Call the <strong>Change Password</strong> dialog box for the currently logged on user.</td>
</tr>
<tr>
<td>$ChangePassword</td>
<td>Call the <strong>Change Password</strong> dialog box for the currently logged on user.</td>
</tr>
</tbody>
</table>

For more information about security functions, see Securing InTouch in the InTouch® HMI Application Management and Extension Guide.

## Specifying and Configuring Users

You can use the following system tag to specify and configure users.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ConfigureUsers</td>
<td>Call the <strong>Configure Users</strong> dialog box.</td>
</tr>
</tbody>
</table>

For more information about security functions, see Securing InTouch in the InTouch® HMI Application Management and Extension Guide.

## Managing Security and Other Information

You can use the following system tags and functions to manage security.

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AccessLevel</td>
<td>Retrieve the access level of the currently logged in user.</td>
</tr>
<tr>
<td>AddPermission()</td>
<td>Assign access levels to a certain user group (local/domain).</td>
</tr>
<tr>
<td>GetAccountStatus()</td>
<td>Retrieve account information (password expiration, lock out, disable flags).</td>
</tr>
<tr>
<td>$InactivityTimeout</td>
<td>Indicate the time that elapses before the user is automatically logged off.</td>
</tr>
<tr>
<td>$InactivityWarning</td>
<td>Indicate the time for the time-out warning.</td>
</tr>
<tr>
<td>InvisibleVerifyCredentials()</td>
<td>Retrieve InTouch access level information of an OS user.</td>
</tr>
</tbody>
</table>
Use | To
--- | ---
IsAssignedRole() | See if the currently logged on user has the specified user role.
QueryGroupMembership() | See if the currently logged on user is a member of a specified user role.

For more information about security functions, see Securing InTouch in the InTouch® HMI Application Management and Extension Guide.

**Miscellaneous Scripting**

InTouch scripting supports sound output so that you can associate human machine interaction with sounds. InTouch scripting also supports getting and setting properties of Wizards.

**Playing Sound Files from an InTouch Application**

In a script, you can associate events and conditions with specific sounds. For example, you could associate a warning dialog box or a critical condition with a warning sound.

**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);
```

**Getting and Setting Properties of Wizards**

Some wizards such as the Distributed Alarm Object and Windows Controls contain set or read properties. These properties could be values in a text box or the check status of a check box.
In a script, you access these properties through the following functions:

<table>
<thead>
<tr>
<th>Use</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetPropertyD(), GetPropertyD()</td>
<td>Set or read discrete properties.</td>
</tr>
<tr>
<td>SetPropertyI(), GetPropertyI()</td>
<td>Set or read integer properties.</td>
</tr>
<tr>
<td>SetPropertyM(), GetPropertyM()</td>
<td>Set or read message properties.</td>
</tr>
</tbody>
</table>

See the wizard description for a list of supported properties.
Here is how to set and read these properties in a generic way.

**GetPropertyD() Function**

Reads a discrete property in a wizard and returns a success code.

**Syntax**

```plaintext
result = GetPropertyD (controlname.property, dtag)
```

**Parameters**

- `controlname`
  The name of a wizard that supports properties. A literal string value, message tagname, or string expression.

- `property`
  The discrete property of the wizard that is to be read. Together with `controlname` can be a literal string value, message tagname, or string expression.

- `dtag`
  The discrete tagname that will receive the discrete property value.

**Return Value**

An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**

With a check box wizard Checkbox1 and a discrete tagname `dtag` you can check the visibility of the check box with the following script function:

```plaintext
result=GetPropertyD("Checkbox1.visible",dtag);
```

This script sets `dtag` to 1, if the check box wizard is visible; otherwise, it sets `dtag` to 0.

**SetPropertyD() Function**

Sets a discrete property in a wizard and returns a success code.

**Syntax**

```plaintext
result = SetPropertyD(controlname.property, Boolean)
```

**Parameters**

- `controlname`
  The name of a wizard that supports properties. A literal string value, message tagname, or string expression.

- `property`
  The discrete property of the wizard that is to be set. Together with `controlname` can be a literal string value, message tagname, or string expression.
**Boolean**
The Boolean value to pass to the wizard property. A literal Boolean value, discrete tagname or Boolean expression.

**Return Value**
An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**
With a check box wizard Checkbox1 and a discrete tagname dtag you can control the visibility of the check box with the following script function:

```plaintext
result=SetPropertyD("Checkbox1.visible",dtag);
```

If you set dtag to 0 and call the script function above, the check box wizard becomes invisible.

**GetPropertyI() Function**
Reads an integer in a wizard and returns a success code.

**Syntax**

```plaintext
result = GetPropertyI (controlname.property, itag)
```

**Parameters**

- `controlname`
The name of a wizard that supports properties. A literal string value, message tagname, or string expression.

- `property`
The integer property of the wizard that is to be read. Together with controlname can be a literal string value, message tagname, or string expression.

- `itag`
The integer tagname that will receive the integer property value.

**Return Value**
An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**
With a radio button wizard Radiobutton1 and an integer tagname itag you can check the currently selected item in the radio button group with the following script function:

```plaintext
result=GetPropertyI("Radiobutton1.value",itag);
```

This script sets itag to 1 (2, 3, ...) , if the first (second, third, ...) radio button is selected.

**SetPropertyI() Function**
Sets an integer property in a wizard and returns a success code.

**Syntax**

```plaintext
result = SetPropertyI (controlname.property, integer)
```

**Parameters**

- `controlname`
The name of a wizard that supports properties. A literal string value, message tagname, or string expression.
**property**
The integer property of the wizard that is to be set. Together with controlname can be a literal string value, message tagname, or string expression.

**integer**
The integer value to pass to the wizard property. A literal integer value, integer tagname, or integer expression.

**Return Value**
An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**
With a radio button wizard Radiobutton1 you can set the 2nd radio button with the following script function:

```plaintext
result=SetPropertyI("Radiobutton1.value",2);
```

**GetPropertyM() Function**
Reads a message property in a wizard and returns a success code.

**Syntax**

```plaintext
result = GetPropertyM (controlname.property, mtag)
```

**Parameters**

- **controlname**
The name of a wizard that supports properties. A literal string value, message tagname, or string expression.

- **property**
The message property of the wizard that is to be read. Together with controlname can be a literal string value, message tagname, or string expression.

- **mtag**
The message tagname that will receive the message property value.

**Return Value**
An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**
With a check box wizard Checkbox1 and a message tagname mtag you can check the caption of the check box with the following script function:

```plaintext
result=GetPropertyM("Checkbox1.caption",mtag);
```

This script sets mtag to the caption of the check box.

**SetPropertyM() Function**
Sets a message property in a wizard and returns a success code.

**Syntax**

```plaintext
result = SetPropertyM (controlname.property, message)
```

**Parameters**

- **controlname**
The name of a wizard that supports properties. A literal string value, message tagname, or string expression.
**property**
The message property of the wizard that is to be set. Together with controlname can be a literal string value, message tagname, or string expression.

**message**
The message value to pass to the wizard property. A literal string value, message tagname, or string expression.

**Return Value**
An integer error code. For more information about the error codes, see Understanding Windows Controls Error Messages in the InTouch® HMI Visualization Guide.

**Example(s)**
With a check box wizard Checkbox1 you can set the caption of the check box wizard dynamically with the following script function:

```plaintext
result = SetPropertyM("Checkbox1.caption","Start Engine 1");
```

This script sets the caption of the check box Checkbox1 to "Start Engine 1".
In This Chapter
About Scripting with OLE Objects ................................................................. 127
Creating, Validating, and Releasing OLE Objects ....................................... 127
Using OLE Object Properties and Methods ................................................ 129
Assigning Multiple Pointers to the Same OLE Object ................................... 130
Troubleshooting OLE Errors ......................................................................... 130
Things You Can Do with OLE ........................................................................ 132

About Scripting with OLE Objects

You can use OLE objects to extend the functionality of an InTouch HMI application. With OLE objects, you can:

- Create popup dialog boxes for the operator interface.
- Access operating system functions, such as the Control Panel.
- Make data from the Manufacturing Execution Module available for processing within the InTouch HMI. See the Manufacturing Execution Module documentation.

Creating, Validating, and Releasing OLE Objects

You can create and validate OLE objects for use in InTouch scripts. After using an OLE object you can release it to free up memory.

Use the following functions to create, validate, and release OLE objects.

- **OLE_CreateObject() Function** on page 127
- **OLE_IsObjectValid() Function** on page 128
- **OLE_ReleaseObject() Function** on page 128

**OLE_CreateObject() Function**

Before you can reference an OLE object in a script, you must create it. When you do this you receive a pointer that references the OLE object.

In a script, you can create an OLE object and assign a pointer by using the OLE_CreateObject() function.

**Syntax**

```c
OLE_CreateObject(%pointer, classname);
```

**Parameters**

- **%pointer**
  The name of your choice for the pointer to the OLE object. It can contain alphanumeric characters (A-Z, 0-9) and underscore. It is case-insensitive.
The name of the OLE class. The class name is case-sensitive. A literal string value, message tagname, or string expression.

**Remarks**

If you use the same object name to create another object, the object is updated to reference the new OLE class. It is released from the old OLE class.

**Example(s)**

This script creates an OLE object called %WShell that references the class Wscript.Shell.

```plaintext
OLE_CreateObject(%WShell, "Wscript.Shell");
```

### OLE_IsObjectValid() Function

In a script, you can verify that an OLE object is valid by using the OLE_IsObjectValid() function. This is not a required step for working with OLE objects, but it is recommended to make sure that you do not come across problems when working with OLE objects.

**Syntax**

```plaintext
result = OLE_IsObjectValid(%pointer)
```

**Arguments**

- `%pointer`
  - The pointer referencing an OLE object that is to be tested.

- `result`
  - A Boolean value indicating the following:
    - 0 - The OLE object the pointer is referencing is invalid.
    - 1 - The OLE object the pointer is referencing is valid.

**Example(s)**

This script creates an OLE object based on the Wscript.Shell class and creates a pointer %WS to reference it. isvalid is a discrete tag that is TRUE if the OLE object is created successfully. Otherwise it is FALSE.

```plaintext
OLE_CreateObject(%WS, "Wscript.Shell");
isvalid = OLE_IsObjectValid(%WS);
```

### OLE_ReleaseObject() Function

After you have used an OLE object in a script, you can release it and delete its pointer to free up system resources. After you release an OLE object you cannot use its pointer to access properties and methods of the associated OLE class.

**Syntax**

```plaintext
OLE_ReleaseObject(%pointer);
```

**Arguments**

- `%pointer`
  - Name of the pointer that references the OLE Object. It can contain alphanumeric characters (A-Z, 0-9) and underscore. It is case-insensitive.

**Example(s)**

This script releases the OLE object associated with the pointer %WShell and deletes the pointer %WShell.

```plaintext
OLE_ReleaseObject(%WShell);
```
Using OLE Object Properties and Methods

In a script, you can use pointers to read and write values from and to OLE properties. You can also use the pointer to call OLE methods. The properties and methods available depend on the OLE object.

Accessing the Properties of an OLE Object

In a script, you can access the properties of an OLE object as you would in most programming languages. Properties are usually identified by using the dot "." operator.

Note: When you use OLE object properties in a script, make sure that their references do not exceed 98 characters, including leading "%". Keep OLE pointer names as short as possible.

Reading an OLE Object Property

In a script, you can read an OLE object property by assigning the property to a tag. You cannot use a direct reference to an OLE object property in an animation display link.

Syntax

tagname = %pointer.property;

Arguments

%pointer
   The pointer that references the OLE object. Must be created with OLE_CreateObject() function or assigned to another pointer before reading a property.

property
   The name of the property to be read.

tagname
   The tag to write the value to.

Example(s)

This script creates an OLE object based on the System.Random OLE class, creates a pointer %SR to reference it, and assigns the value of the .NextDouble property of the Math.Random OLE object to a real tagname randtag.

At run time the real tagname Randtag receives a random double float value between 0 and 1.

```
OLE_CreateObject(%SR,"System.Random");
randtag = %SR.NextDouble;
```

Writing to an OLE Object Property

In a script, you can write a value to an OLE object property by assigning a value to the property.

Syntax

%pointer.property = value;

Arguments

%pointer
   The pointer that references the OLE object. Must be created with OLE_CreateObject() function or assigned to another pointer before writing to a property.

property
   The name of the property to be written to.

value
   The value to be written to the property. It can be a literal value, tagname or expression. Writing to an OLE property from an animation input link directly is not supported.
Calling Methods of an OLE Object

In a script, you can call OLE object methods.

**Syntax**

```
%pointer.method(parameters);
```

**Arguments**

- **%pointer**
  The pointer that references the OLE object. Must be created with OLE_CreateObject() function or assigned to another pointer before calling a method.

- **method**
  The name of the method that is part of the OLE object.

- **parameters**
  A list of parameters to pass to the method. These parameters must be separated by comma. Literal values, tag names or expressions.

**Example(s)**

This script creates an OLE object based on the OLE class Shell.Application, creates a pointer %sa to the OLE object and calls the method .MinimizeAll(). This method minimizes all windows on your desktop.

```
OLE_CreateObject(%SA,"Shell.Application");
%SA.MinimizeAll();
```

**Note:** Optional parameters are not allowed in OLE InTouch HMI scripting. All parameters must be specified.

Assigning Multiple Pointers to the Same OLE Object

In a script, you can assign multiple pointers to the same OLE object by using the equals sign.

**Syntax**

```
%newpointer = %pointer
```

**Arguments**

- **%pointer**
  The name of the pointer that already references a created OLE object.

- **%newpointer**
  The name of a new pointer that should reference the same OLE object. It can contain alphanumeric characters (A-Z, 0-9) and underscore. It is case-insensitive.

**Example(s)**

This script creates an OLE object based on the Wscript.Shell class and creates a pointer %WS to reference it. The pointer %WS2 when set to %WS points to the same OLE object. It can be used to read from or write to properties and call methods of the same OLE object.

```
OLE_CreateObject(%WS,"Wscript.Shell");
%WS2=%WS;
```

**Note:** You can use message tagnames in connection with pointers. If you assign a message tagname to a pointer, it can get an ID value. You can use it to create more pointers to the same OLE object.

Troubleshooting OLE Errors

In a script, you can use OLE functions to troubleshoot OLE errors.
### OLE_GetLastErrorObject() Function

This function returns the error number of the last OLE error.

**Syntax**

```plaintext
errnum = OLE_GetLastErrorObject();
```

**Arguments**

- `errnum`
  - The number of the last OLE error.

### OLE_GetLastErrorObjectErrorMessage() Function

This function returns the error message of the last OLE error.

**Syntax**

```plaintext
errmsg = OLE_GetLastErrorObjectErrorMessage();
```

**Arguments**

- `errmsg`
  - The error message of the last OLE error.

### OLE_ResetLastErrorObject() Function

In a script, use the OLE_ResetLastErrorObject() function to reset the last OLE error so that the last OLE error number is set to zero and last OLE error message is set to blank.

This can be used for identifying any errors in a batch of OLE functions.

**Syntax**

```plaintext
OLE_ResetLastErrorObject();
```

### OLE_ShowMessageOnLastErrorObject() Function

By default, when an OLE error occurs, an error message dialog box is displayed.

In a script, you can specify whether or not to display the error message dialog box by using the function OLE_ShowMessageOnLastErrorObject().
**Syntax**

```
OLE_ShowMessageOnObjectError(Boolean)
```

**Arguments**

```
Boolean
```

A value that determines if an OLE error message dialog box is displayed or not. A literal Boolean
value, discrete tagname or Boolean expression with following meanings:

- **0** - no OLE error message dialog box is displayed when an OLE error occurs
- **1** - an OLE error message dialog box is displayed when an OLE error occurs

**Example(s)**

This script suppresses all OLE error message dialog boxes. When OLE errors occur, no error message
dialog boxes are displayed.

```
OLE_ShowMessageOnObjectError(0);
```

**OLE_IncrementOnObjectError() Function**

In a script, you can use the OLE_IncrementOnObjectError() function to designate an integer tagname as
counter for the number of OLE errors.

**Syntax**

```
OLE_IncrementOnObjectError(integertag)
```

**Parameters**

```
integertag
```

The tagname that acts as a counter.

**Remarks**

If OLE error message dialog boxes are displayed, the counter tagname is only incremented after the
OLE error message dialog box is closed.

**Example(s)**

This script designates the integer tagname errorcount as error counter, hides the error message dialog
boxes and attempts to create an OLE object based on an invalid OLE class name. This creates an error
and the tagname value errorcount is incremented to 1.

```
errorcount = 0;
OLE_IncrementOnObjectError(errorcount);
OLE_ShowMessageOnObjectError(0);
OLE_CreateObject(%WS,"InVaLiD.cLaSs.nAmE");
```

**Things You Can Do with OLE**

You can use the following scripts to get an idea of the powerful functionality you can add to an application
using OLE objects.

**Produce Random Numbers**

In a script, use the following commands to produce a random number between 0 and 255:

```
OLE_CreateObject(%SR,"System.Random");
randtag = (%SR.NextDouble)*255;
```

**Create User Interface Dialog Boxes**

In a script, use the following commands to produce a user interface dialog box:

```
dim DlgBody as message;
dim DlgTitle as message;
```
dim Style as integer;
dim Result as integer;
DlgBody = "Do you want to open the valve 'MR-3-FF'?";
DlgTitle = "Confirm Opening Valve MR-3-FF";
Style = 48;
OLE_CreateObject(%WS,"Wscript.Shell");
result = %WS.Popup(DlgBody,1,DlgTitle,Style);

This example creates the following user interface dialog box.

![Confirm Opening Valve MR-3-FF dialog box](image)

The Style tagname determines which icon and which buttons appear on the dialog box. Use the following values:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Style</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(no icon)</td>
<td>no icon</td>
<td>0</td>
</tr>
<tr>
<td><img src="image" alt="Error icon" /></td>
<td>Error icon</td>
<td>16</td>
</tr>
<tr>
<td><img src="image" alt="Question mark icon" /></td>
<td>Question mark icon</td>
<td>32</td>
</tr>
<tr>
<td><img src="image" alt="Warning icon" /></td>
<td>Warning icon</td>
<td>48</td>
</tr>
<tr>
<td><img src="image" alt="Information icon" /></td>
<td>Information icon</td>
<td>64</td>
</tr>
</tbody>
</table>

To use a particular button, add one of the following values to the Style value:

<table>
<thead>
<tr>
<th>Value</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Only <strong>OK</strong> button</td>
</tr>
<tr>
<td>1</td>
<td><strong>OK</strong> and <strong>Cancel</strong> buttons</td>
</tr>
<tr>
<td>2</td>
<td><strong>Abort</strong>, <strong>Retry</strong> and <strong>Ignore</strong> buttons</td>
</tr>
<tr>
<td>3</td>
<td><strong>Yes</strong>, <strong>No</strong> and <strong>Cancel</strong> buttons</td>
</tr>
<tr>
<td>4</td>
<td><strong>Yes</strong> and <strong>No</strong> buttons</td>
</tr>
<tr>
<td>5</td>
<td><strong>Retry</strong> and <strong>Cancel</strong> buttons</td>
</tr>
</tbody>
</table>
Scripting with OLE Objects

InTouch HMI Scripting and Logic Guide

Value | Style
-------|--------
6 | Cancel, Try Again and Continue buttons

The Result tagname contains the button number the user clicked. This can be used for conditional branching in your InTouch script. Following result codes are possible:

<table>
<thead>
<tr>
<th>Result Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OK button was pressed</td>
</tr>
<tr>
<td>2</td>
<td>Cancel button was pressed</td>
</tr>
<tr>
<td>3</td>
<td>Abort button was pressed</td>
</tr>
<tr>
<td>4</td>
<td>Retry button was pressed</td>
</tr>
<tr>
<td>5</td>
<td>Ignore button was pressed</td>
</tr>
<tr>
<td>6</td>
<td>Yes button was pressed</td>
</tr>
<tr>
<td>7</td>
<td>No button was pressed</td>
</tr>
<tr>
<td>10</td>
<td>Try again button was pressed</td>
</tr>
<tr>
<td>11</td>
<td>Continue button was pressed</td>
</tr>
</tbody>
</table>

Open Windows Date and Time Properties Panel

In a script, use the following commands to open the Windows Date/Time Properties panel:

```
OLE_CreateObject(%WP,"Shell.Application");
%WP.SetTime();
```

You can do similar tasks by calling different methods and passing them to the referenced OLE object:

<table>
<thead>
<tr>
<th>This Method</th>
<th>Opens The Panel</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrayProperties()</td>
<td>Tray properties</td>
</tr>
<tr>
<td>FileRun()</td>
<td>File Run dialog box</td>
</tr>
<tr>
<td>FindFiles()</td>
<td>Find Files dialog box</td>
</tr>
<tr>
<td>FindComputer()</td>
<td>Find Computer dialog box</td>
</tr>
<tr>
<td>ShutdownWindows()</td>
<td>Shutdown Windows panel</td>
</tr>
</tbody>
</table>

Read and Write to the Registry

In a script, you can use OLE to read from and write to the Windows registry by:

- Creating an OLE object based on the Windows class Wscript.Shell.
- Using the RegRead() and RegWrite() methods of the OLE object.
For example, these commands read the installed version of the InTouch HMI directly from the registry key and store the value in the rkey message tagname:

```
OLE_CreateObject(%WS,"Wscript.Shell");
rkey = %WS.RegRead("HKLM\SOFTWARE\Wonderware\InTouch\Installation\Version");
```

These commands write the value 1 to the registry key that determines if file extensions are hidden for the currently logged on user:

```
OLE_CreateObject(%WS,"Wscript.Shell");
%WS.RegWrite("HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\HideFileExt",1,"REG_DWORD");
```

## Minimize Windows

In a script, you can use the following commands to minimize all windows on your desktop:

```
OLE_CreateObject(%WA,"Shell.Application");
%WA.MinimizeAll();
```

You can do similar tasks by calling these methods:

<table>
<thead>
<tr>
<th>This Method</th>
<th>Arranges Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>TileHorizontally()</td>
<td>Tiles all Windows horizontally</td>
</tr>
<tr>
<td>TileVertically()</td>
<td>Tiles all Windows vertically</td>
</tr>
<tr>
<td>CascadeWindows()</td>
<td>Cascades all Windows</td>
</tr>
<tr>
<td>UndoMinimizeALL()</td>
<td>Restores all Windows</td>
</tr>
</tbody>
</table>
You can use ActiveX controls to read from and write to tag names and I/O references. In a script, you can reference ActiveX controls.

You can also create scripts that execute when an event occurs for the ActiveX control. These scripts can be re-used and imported into other applications.

In This Chapter

About Scripting ActiveX Controls ................................................................. 137
Calling ActiveX Control Methods ................................................................. 137
Accessing ActiveX Control Properties from the InTouch HMI .......................... 138
Creating and Re-using ActiveX Event Scripts .............................................. 140
Importing ActiveX Event Scripts ................................................................. 143

About Scripting ActiveX Controls

You can use ActiveX controls to read from and write to tag names and I/O references. In a script, you can reference ActiveX controls.

You can also create scripts that execute when an event occurs for the ActiveX control. These scripts can be re-used and imported into other applications.

Calling ActiveX Control Methods

In a script, you can call methods of an ActiveX control to perform actions supported by the ActiveX control. ActiveX methods can be called from any type of InTouch QuickScript or ActiveX Event script.

Note: To call the ActiveX method when an ActiveX event occurs, there are some prerequisite things you need to do. See Configuring ActiveX Event Scripts on page 39.

To call an ActiveX control method

1. In a script dialog box, on the Insert menu, click ActiveX.
The **ActiveX Control Browser** dialog box appears.

2. Click the name of the ActiveX control from the left pane. The right pane contains the names of properties and methods that are supported by the ActiveX control.

3. Click the name of the method to use from the right pane and then click **OK**. The method name and default parameters are pasted into the script window at the cursor position.

4. Configure the method parameters inside the parentheses, to your specifications.

### Accessing ActiveX Control Properties from the InTouch HMI

In a script, you can read from and write to ActiveX control properties to exchange data between the ActiveX control and the InTouch tagnames and display links.

#### Configuring ActiveX Control Properties to Read and Write Data

In a script, you can read data from and write data to an ActiveX control. You use the ActiveX control properties associated with specific ActiveX controls.

There are two ways of doing this:

- Use the ActiveX control property in an InTouch HMI QuickScript or ActiveX event script. The property value is read or written every time the script is executed.

- Link the ActiveX control property directly to an InTouch HMI tag or I/O reference. The property value is read or written at every update interval.

### Configuring Scripts to Read and Write ActiveX Control Properties

In a script, you can configure ActiveX control properties to either write values to or read values from InTouch HMI tagnames or other expressions.
To read data from or write data to an ActiveX control property

1. Open a script window, point to Insert, and click ActiveX. The ActiveX Control Browser dialog box appears.

2. Click the name of the ActiveX control from the left pane. The right pane contains the names of properties and methods of the selected ActiveX control.

3. Click the name of the property to use from the right pane. The property name is inserted into the script window at the cursor position.

4. Assign the property name to a tag or use according to your specifications.

5. Click OK.

Example(s)

The following script reads the ToPriority property of the ActiveX control instance AlarmViewerCtrl1 into the integer tagname topri.

```plaintext
topri = #AlarmViewerCtrl1.ToPriority;
```

The following script writes the value MS Comic to the Font property of the ActiveX control called AlarmViewerCtrl1. This example changes the display font of the AlarmViewer ActiveX control dynamically.

```plaintext
#AlarmViewerCtrl1.Font = "MS Comic";
```

**Linking ActiveX Control Properties to Tag or I/O References**

You can link ActiveX control properties to InTouch HMI tags or I/O references.
To link ActiveX control properties to tags or I/O references
1. Double-click the ActiveX control. The properties dialog box of the ActiveX control appears.

![ActiveX Control Properties Dialog Box]

2. Click the **Properties** tab and scroll to the right.
3. Select the property in the list.

![Properties Tab and List]

4. Assign a tag or I/O reference. Do either of the following:
   - Type the tag or I/O reference directly into the **Associated Tag** column.
   - Click the ellipsis button in the **Associated Tag** column between the square parenthesis. The **Select Tag** dialog box appears. Select a tag and click **OK**.
5. Click **OK**.

Creating and Re-using ActiveX Event Scripts

An ActiveX control can support events, such as single-clicking on the control, that you can use to associate certain actions with. These actions are stored in ActiveX event scripts.

Creating ActiveX Event Scripts

You can create or re-use an event script that is executed every time a specific ActiveX control event occurs, such as clicking on an ActiveX control.

**To create an ActiveX event script**
1. Double-click the ActiveX control. The properties dialog box appears.
2. Click the **Events** tab.

3. Click the event to associate. Brackets and ellipses appear in the **Script** column.

4. In the **Script** column of the corresponding row, click between the brackets.

5. Enter a new name and click **OK**. When a message appears, click **OK**. The **ActiveX Event Scripts** dialog box appears.

6. Create the script according to your specifications.

**Re-using ActiveX Event Scripts**

You can re-use ActiveX event scripts if they are created by the same ActiveX control parent and event. For example, if you have multiple AlarmViewer ActiveX controls in an application, they can share event scripts for the *DoubleClick* event.

To re-use an ActiveX event script

1. Double-click the ActiveX control. The properties dialog box appears.

2. Click the **Events** tab.
3. Click the event to associate. Brackets and ellipses appear in the **Script** column.

![ActiveX Event Script Dialog Box](image)

4. In the **Script** column of the corresponding row, click the ellipsis button. The **Choose ActiveX Script** dialog box appears.

5. Click an ActiveX script and click **OK**.

6. Click **OK** again.

### Creating Self-Referencing ActiveX Event Scripts

If you use ActiveX event scripts, you can configure them to reference themselves instead of an absolute ActiveX control name. This is useful when you create ActiveX event scripts that will be re-used. ActiveX event scripts can either:

- Reference the specific ActiveX control that produced the event (**ThisControl**).
- Reference the specific event that called the script (**ThisEvent**).

Referencing the specific event enables the ActiveX control to pass other parameters to the ActiveX control script.

**To create self-referencing ActiveX event scripts**

1. Create an ActiveX event script for a specific ActiveX event. See *Creating ActiveX Event Scripts* on page 140.

2. In the **ActiveX Event Script** dialog box, click **Insert**, and then click **ActiveX**. The **ActiveX Control Browser** dialog box appears.

![ActiveX Control Browser](image)

3. In the left pane, do one of the following:
   - Click **ThisControl** to see properties and methods that you can use in connection with this control (and any other control that you re-use this script in).
Click **ThisEvent** to see properties and methods of the ActiveX control that you can use in connection with the self-referencing event.

4. In the right pane, click one of the properties or methods and click **OK**. The selected property or method is pasted to the script window.

5. Configure the script.

6. Click **OK**.

   For example, this statement writes the value of the ClicknRow event parameter to the ClickedRow tag:

   ```
   ClickedRow = ThisEvent.ClicknRow;
   ```

### Importing ActiveX Event Scripts

You can import ActiveX event scripts from other InTouch HMI applications so as to re-use them in the application currently under development.

**To import ActiveX event scripts from other applications**

1. On the **File** menu, click **Import**. The **Import from directory** dialog box appears.
2. Browse to the InTouch HMI application that contains the ActiveX event scripts to import.
3. Click **OK**. The **Application Data Import Options** dialog box appears.

   ![Application Data Import Options]

4. Select the **ActiveX Event Scripts** check box and click **Import**. All ActiveX event scripts are imported into the current InTouch HMI application.
Troubleshooting QuickScripts

You can troubleshoot QuickScripts by using the Log Viewer to display run time values of tagnames.

Logging Messages to the Log Viewer

LogMessage(messagetag);  // In this script, messagetag is the name of a message tagname whose value you want to log.
LogMessage(StringFromIntg(inttag,10));  // In this script, inttag is the name of an integer tagname whose value you want to log.
LogMessage(Text(realtag,"#.#######"));  // In this script, realtag is the name of a real tagname whose value you want to log.
LogMessage(DText(disctag,"TRUE","FALSE"));  // In this script, disctag is the name of a discrete tagname whose value you want to log.
LogMessage("DEBUG tag:""+ind.name+" value:"+Text(ind,"#.####"));  // In this script, ind could be an analog indirect tag.

LogMessage() Function

LogMessage("Message Tag");
Parameter

Message_Tag
String to log to the Log Viewer. Actual string or message tagname.

Remarks
This is a very powerful function for troubleshooting InTouch scripting. By strategically placing LogMessage() functions in your scripts, you can determine the order of QuickScript execution, the performance of scripts, and identify the value of tags both before they are changed and after they have been affected by the QuickScript. Each message posted to the Log Viewer is stamped with the exact date and time.

Important: The percent (%) character formats diagnostic messages that appear in the SMC Log Viewer while debugging scripts. WindowViewer can stop responding if the % character appears in a log string or a function parameter. To eliminate errors caused by %, use two %% characters.

Example(s)
LogMessage("Report Script is Running");

The above statement would print the following to the Log Viewer:
LogMessage("The Value of MyTag is " + Text(MyTag, ")");
MyTag = MyTag + 10;
LogMessage("The Value of MyTag is " + Text(MyTag, ")");

Viewing Log Viewer Messages

The Log Viewer is located in the ArchestrA System Management Console (SMC) and is installed when you install the InTouch HMI.

To view the logged values in Log Viewer
1. Click Start, point to Programs, point to Wonderware, and then click ArchestrA System Management Console. The ArchestrA System Management Console appears.
2. In the left pane expand **Log Viewer**, expand **Default Group**, and then click **Local**. The Log Viewer messages appear in the details pane.

![Log Viewer Screenshot](image)

3. Locate the logged values from the LogMessage() function.

**Note:** If you are debugging the script on a remote InTouch HMI node, you must add the Node name to the Node Group in Log Viewer and view the Log Viewer messages of that node.
Index

A
action scripts
  opening IXactionscriptsopening • 17
  triggers IXactionscriptstriggers • 30
ActiveX
  calling a method • 137
  control browser • 137
  inserting methods or properties in a script • 26
  linking properties to tags • 139
ActiveX event scripts
  creating IXActiveXeventscreating • 140
  importing IXActiveXeventsimporting • 143
  opening IXActiveXeventsopening • 17
  re-using IXActiveXeventsreusing • 141
  self-referencing IXActiveXeventsselfreferencing • 142
  triggers IXActiveXevents triggering • 30
AddPermission() function • 120
application scripts
  opening IXapplicationsopening • 17
  triggers IXapplications triggering • 30
AttemptInvisibleLogon() function • 119

B
branching
  IF-THEN-ELSE • 55

C
ChangePassword() function • 120
comments in scripts IXcommentsinscripts • 44
condition scripts
  triggers IXconditionscriptstriggers • 30
converting
  discrete value to string • 84
  integer value to string • 82
string to integer value • 83
string to real value • 84
creating
scripts • 13

D
data change scripts
  triggers • 30
discrete values, converting to string • 84
dot fields, inserting • 26
DText() function • 84

E
editing scripts • 13
editor • 14

F
functions
  definition • 11
  OLE_CreateObject() function • 127
  OLE_GetLastObjectError() function • 131
  OLE_GetLastObjectErrorMessage() function • 131
  OLE_IsObjectValid() function • 128
  OLE_ReleaseObject() function • 128
  OLE_ResetObjectError() function • 131
  OLE_ShowMessageOnObjectError() function • 131

G
GetAccountStatus() function • 120

I
indenting script statements • 44
inserting
dot fields • 26
inserting functions • 26
inserting tag names • 26
integers, converting from string • 83
integers, converting to string • 82
InvisibleVerifyCredentials() function • 120
IsAssignedRole() function • 120
K
key scripts
  triggers IX keyscriptstriggers • 30

L
literal data values IX literal datavalues • 44
Log Viewer IX LogViewer • 146
LogMessage() function IX LogMessage function • 145
LogOff() function • 119
LogonCurrentUser() function • 119

M
MEM OLE • 14
multiple triggers IX multiple triggers • 30

O
OLE
  counting error messages IX OLE counting error messages • 132
OLE objects
  reading a property • 129
  validity • 128
  writing a property • 129
OLE_CreateObject() function • 127
OLE_GetLastObjectError() function • 131
OLE_GetLastErrorObjectErrorMessage() function • 131
OLE_IsObjectValid() function • 128
OLE_ReleaseObject() function • 128
OLE_ResetObjectError() function • 131
OLE_ShowMessageOnObjectError() function • 131

P
periodic script execution IX periodic script execution • 30
PostLogonDialog() function • 119

Q
QueryGroupMembership() function • 120
QuickFunctions • 61
  calling • 62
  configuring • 61
  creating • 61
  definition • 11
window scripts
  opening • 17
  triggers • 30
windows
  moving and resizing • 89